Handbook for the Computer Security Certification of Trusted Systems

NRL Technical Memorandum 5540:133A, 31 Mar 1995

Naval Research Laboratory, Code 5540, Washington, D.C. 20375-5337

For additional copies of this report, please send e-mail to
landwehr@itd.nrl.navy.mil, or retrieve PostScript via
http://www.itd.nrl.navy.mil/ITD/5540/publications/handbook
(e.g., using Mosaic).

IMPLEMENTATION EVALUATION GUIDELINE

A Chapter of the

Handbook for the Computer Security Certification

of

Trusted Systems

In Support of
AFSEC DO #5120

Prepared for:
Naval Research Laboratory
Washington, D. C.

Prepared by:
CTA INCORPORATED
7150 Campus Drive, Suite 100
Colorado Springs, CO 80920

Table of Contents

IMPLEMENTATION EVALUATION GUIDELINE

FOREWORD

This report is one chapter of the Naval Research Laboratory (NRL) Handbook entitled *The Handbook for the Computer Security Certification of Trusted Systems*. This chapter addresses the security evaluation of the implementation phase of a trusted application system development effort and its results, particularly the software results. The purpose and structure of the handbook are described in [9].

This chapter's primary audience is system security evaluators. The chapter provides these evaluators an approach for assessing an implementation of a trusted application system. The approach is relatively independent of the particular development method used to produce the implementation. System developers, including managers and procurement agencies, form a secondary audience. The chapter provides these developers a guideline for producing an implementation that lowers the risk of failing certification at the B3 level of assurance. [23]

The definition and scope of the term "implementation phase" within the development life-cycle is provided in Section 1. Important differences between a trusted implementation and an untrusted one are identified in Section 2, along with supporting rationale. It is important for an evaluation team to assess the product(s) of the implementation phase, termed work products in this chapter, and to also assess the process that produces those products. Elements that distinguish acceptable work products of the implementation phase and those elements that make an acceptable implementation process are discussed in Section 3 and Section 4, respectively. Section 5 presents a suggested method by which an evalation team can assess an implementation. Section 6 provides, in accordance with the suggested method, an example of the evaluation results of an actual system.

This chapter assumes that a design assessment has been successfully completed and that the necessary documentation has been produced by the development and the design evaluation teams. This chapter assumes that the implementation evaluation team has access to an adequate set of design and design assessment documents for the system being assessed. This chapter further assumes that the design documentation has already been accepted by the appropriate authority as satisfactorily meeting all B3 design requirements, which include a software development plan (SDP), updated to reflect information obtained during the design assessment phase. This chapter also assumes that the implementation evaluation team may be different than the design evaluation team.

The focus of this chapter, as is the focus of the entire handbook, is on the B3 level of assurance and on systems as opposed to products. The distinction between "system" versus "product," however, is not significant for evaluating an implementation[1]. The distinction may be a significant factor for the evaluation of the security policy modeling, the associated descriptive top-level specifications, or the design of a trusted application system. Such issues, however, will have been worked out by the implementation phase. Therefore, there is no emphasis in this chapter on the distinction between "system" versus "product."

This chapter focuses on guidelines for an assessment of the software of a trusted application system. Although there is a software focus, the principles presented here do carry over into an assessment of the underlying hardware. This is compatible with the requirements of the *Trusted Computer Security Evaluation Criteria* (TCSEC) [23], which specifies that the trusted computing base (TCB), as well as the system, is composed of

[1]As defined in the *Information Technology Security Evaluation Criteria* (ITSEC) [5] and following NRL guidance [9], a *system* is a specific installation "with a particular purpose and a known operational environment." A *product*, on the other hand, is a "hardware and/or software package that can be bought off the shelf and incorporated into a variety of systems." A *trusted system* is a system that has been certified against some trust criteria.

hardware, firmware, and software. Although the TCSEC indicates the necessity of certain hardware features, it barely references the hardware design and does not levy any direct requirements on a hardware implementation. Another reason for the software focus is that the primary focus of this chapter is on the implementation phase, which occurs before the system integration and testing phase that brings together in the development process, both the hardware and software. This latter aspect is discussed further in the next section where the implementation phase is placed within the DoD-STD-2167A [7] development context.

1 WHAT IS MEANT BY AN IMPLEMENTATION?

This section presents the context of the evaluation guideline in terms of the system life cycle. This chapter acknowledges that the distinctions among design, implementation and system integration may not be well defined, particularly for projects that integrate COTS products or use an evolutionary development method.

1.1 Implementation Phase Defined

The focus of this chapter is on the implementation phase of the software development life cycle. By the term *implementation phase*, we mean that portion of the development life cycle that starts with a successful completion of a critical design review (CDR), includes the production of source code, the successful unit testing, computer software component (CSC) integration and testing, computer software configuration item (CSCI) testing, and finishes with acceptance into the system integration and test phase. As identified in ANSI/IEEE Std 729-1983 [28], it is the period of time in the system life cycle during which the components of a software product are created from design documentation and debugged. As described in [28], an *implementation* is a realization of an abstraction in more concrete terms; in particular, in terms of hardware, software, or both. An implementation is also described as a machine executable form of a program or a form of a program that can be translated automatically to machine executable form.

Specifically, for this chapter, the implementation phase also includes the software integration activity, which involves combining software elements, into an overall system. As identified in DOD-STD-2167A [7], this is the period of time during which the CSCs are integrated and tested and the CSCI testing is accomplished. The source code listings are incorporated into the developmental configuration within this phase. Updated source code and software test reports are products of the integration process. Software product specifications may be a product of this phase although they may be deferred until after system integration and testing. The process of testing an integrated hardware and software system to verify that the integrated system meets its specified requirements occurs, however, during the *system* integration and testing phase. Therefore, this latter activity occurs after this chapter's definition of software integration. Explicit security testing and penetration testing also occur outside of this chapter's definition of software integration.

These relationships are identified in Figure 1-1, which is taken directly from [7]. Note that the figure incorporates a brace indicating where, for this chapter's discussion, the implementation phase is identified and placed in relation to the development process.

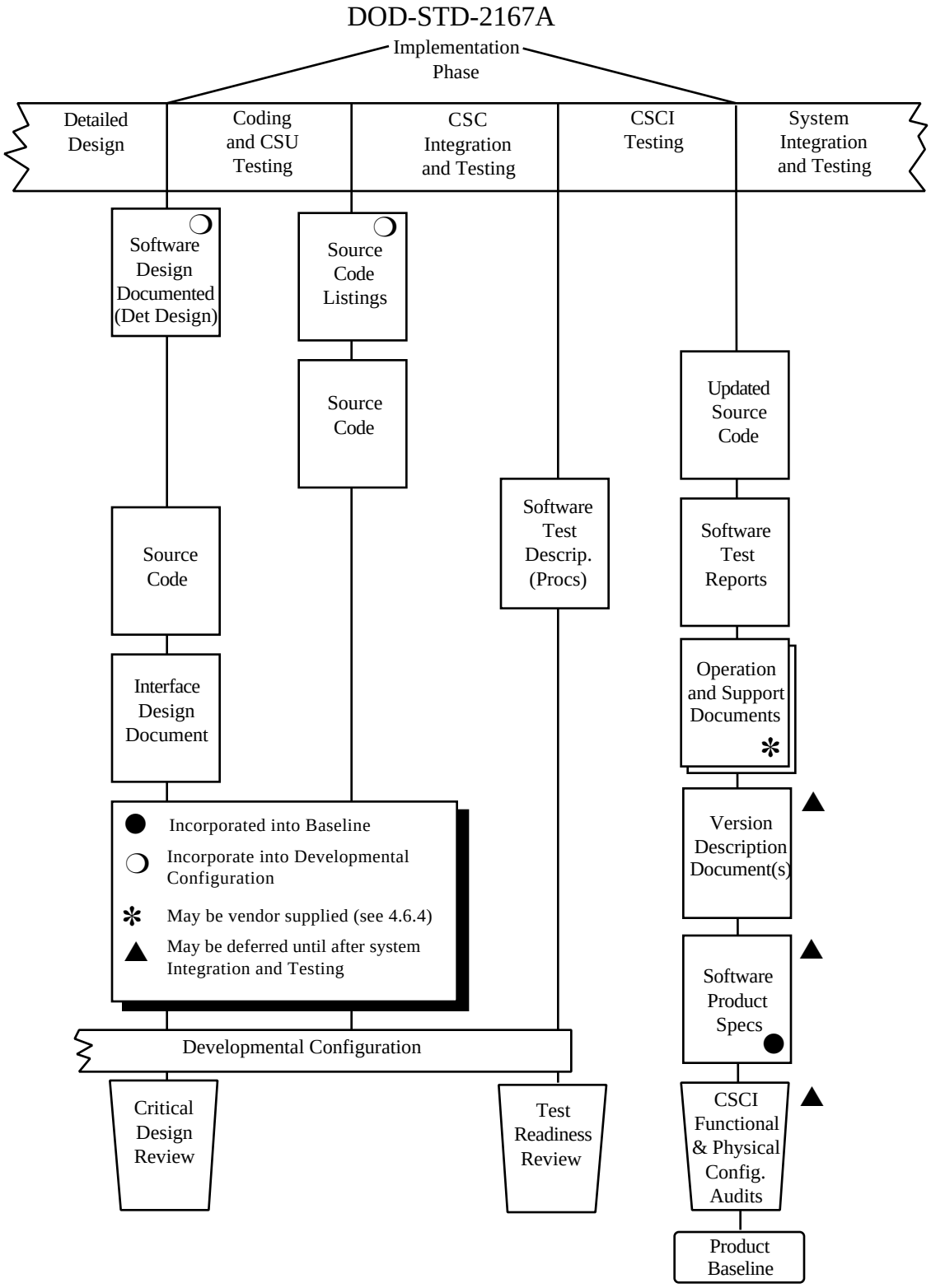

Figure 1-1. 2167A Deliverable products, reviews, audits and baselines (portion of).

1.2 Relationship to Life Cycle Models

The life cycle model that has most generally been used in past development efforts has been the traditional "waterfall model." Such a model tends to imply that all activities of one phase or stage must be completed before starting the next phase or stage.

Although the information presented in this chapter is presented in the context of a waterfall development method, it is essential that an evaluator understand that development approaches other than a traditional approach may be well suited for achieving the B3 goal. Such approaches may be encouraged if an approach can provide the necessary insight and understanding to a third party assessment team. It is incumbent on the developer, however, to provide the rationale that would meet the assessment requirements. A goal of this chapter is to identify the fundamental principles and issues so that one can obtain a proper understanding of an implementation and its integration. This chapter provides input into those assessment requirements.

If prototyping or modular development is to be used for a project where different modules may be developed and completed at significantly different times, each module can be considered to evolve through the phases and stages of the "waterfall model" independent of the other modules. Management control must be used so that proper software engineering is applied to ensure that all requirements are satisfied and that proper testing for the integration of all modules is planned and executed. Planning for the documentation is essential to ensure that available information is recorded since the use of modular development may dictate that only portions of documents be written pending completion of the related modules and their documentation.

An example of such an approach is "Evolutionary Rapid Prototyping," which is described in [6]. The development concept is to provide an easily modifiable and extensible working model of a proposed system that is not necessarily representative of the complete system. The model provides users of the application with a physical representation of key parts of the system before implementation. The goal of such an approach is to evolve the prototype into the final system, to allow an effective means to communicate the user interface definition, and to allow user input throughout the development process. How the design and implementation information is presented and coordinated with a certification team will need to be determined by the developer in conjunction with the security evaluation team. Although the principles are identified upon which a determination can be based, the actual determination is beyond the scope of this chapter.

2 WHAT IS THE DIFFERENCE BETWEEN A TRUSTED IMPLEMENTATION AND AN UNTRUSTED ONE?

Development of a trusted application system is similar in many ways to the development of an untrusted system to DoD or MIL Standards. As an example of the similarity, the paper, "Developing Trusted Systems using 2167A" [30] gives an approach to tailoring DoD-STD-2167A data item descriptions to incorporate TCSEC requirements. Despite the similarity, there are differences.

- More attention is given to preventing extraneous code or extra functionality for trusted systems.
- Specific emphasis is placed on the verification of assertions. The verification of assertions that can be made about other software or hardware or environmental or operational conditions continues from the previous phases of the development life cycle.
- An emphasis is placed on assurance through understanding of the implementation by an independent evaluation team.
- An emphasis is placed on access to information. The evaluation team may want to access the informal design and implementation documentation beyond that which is delivered normally under contract, e.g. software development folders or programmers notebooks. The intent is access --- not to make extra work for the developer or make such information a formal delivery.
- The testing of high assurance systems goes beyond the standard testing that demonstrates system functionality. The testing philosophy is to reduce the risk in the release by testing to reveal potential defects. The difference in philosophy is illustrated by the analogy given in [2] of the difference between bookkeepers and auditors. A bookkeeper's goal is to show that the books balance, but an auditor's goal is to determine if the bookkeeper has embezzled despite the appearance of balance. This difference in testing philosophy leads to a more methodical approach to the testing activities. It tends to lead to more rigorous tests developed by testing specialists.

These differences stem from the requirements for trusted systems including the requirements on any approach to produce such high assurance systems. These aspects are discussed next. The remainder of this chapter provides additional rationale for each of these stated differences.

Finally, these differences identify, in effect, risk areas in achieving a B3 implementation. Paying attention to these areas, a trusted application system development program will reduce the risk of certification failure at B3 level.

2.1 Implementation Related B3 Requirements

An analysis of B3 requirements shows that there are two direct requirements and one derived requirement levied by the TCSEC on an implementation. There are, however, a number of implementation constraints related to the B3 design requirements that are important to the support of B3 assurance. This chapter discusses those as well. Aspects of B3 assurance not included in the B2 class are distinguished.

One direct requirement (Security Testing) is:

> *No design flaws and no more than a few correctable implementation flaws may be found during testing, and there shall be reasonable confidence that few remain.*

This is a direct requirement on the implementation.

The second direct requirement (Configuration Management) is:

> *During development and maintenance of the TCB, a configuration management system shall be in place that maintains control of changes to . . . implementation documentation, source code, the running version of the object code, and test fixtures and documentation . . . . Tools shall be provided for generation of a new version of the TCB from the source code. Also available shall be tools for comparing a newly generated version with the previous TCB version in order to ascertain that only the intended changes have been made in the code that will actually be used as the new version of the TCB.*

This is a direct requirement on the implementation.

The derived requirement (Design Documentation) is:

> *The TCB implementation . . . shall be informally shown to be consistent with the DTLS. The elements of the DTLS shall be shown, using informal techniques, to correspond to elements of the TCB.*

This is not a direct requirement because no requirement regarding the implementation is specifically stated. Nevertheless, such a consistency and correspondence demonstration will not be possible unless the implementation obeys certain constraints. As an example of a constraint on an implementation due to a design requirement, consider the modularity requirement (System Architecture):

> *The TCB shall be internally structured into well-defined largely independent modules.*

This is a design requirement. If this is not carried over into the implementation as a constraint, then the correspondence or mapping requirement from the TCB implementation to the descriptive top-level specification (DTLS) will not be satisfied. Note that there appears to be no need for an additional derived requirement to be explicitly identified and levied on the implementation to assure the modularity of the implementation.

2.2 Assurance Through Understanding

The fundamental theme that an evaluation team needs to follow is that of obtaining *assurance through understanding.* All products assessed using the TCSEC have an associated trusted computing base. How well that TCB is identified and structured, together with its functionality with respect to the rest of the system, is fundamental to an assessment of its assurance. A key element of the assurance at the higher assurance levels, in particular at B3, is the direct relationship between the *level of understanding a third party can obtain* regarding the system's design and its implementation and the *assurance achieved*. It is necessary to have a third party or evaluation team reach the conclusion that the requirements are satisfied based on its own analysis and testing. The effort will be accomplished, no doubt, in concert with the development team. It is not sufficient that the development team believes it met the requirements. It is the evaluation team's conclusion that is important.

There are three aspects that contribute to the achievement of the assurance through understanding. They are:

- Ensuring that the implementation is a faithful realization of the design,
- The clarity of the implementation on its own, and
- The planning, execution and documentation of the implementation process.

An implementation is a faithful realization of a design when it has been determined that all aspects specified in the design and assessed in the design evaluation are carried over into the implementation in an understandable manner. Examples include the processing structure, the functional realization of design objects, and the realization of interfaces. The clarity of an implementation involves those aspects of the implementation that may not have been fully specified or assessed during the design evaluation process, yet may impact the first area. Finally, one obtains assurance through assessing how well the implementation phase has been planned, executed and documented. It is important that there has been a solid software engineering process in place to produce the product. Each of these three aspects is discussed in further detail in the following sections.

2.3 Rationale for Evaluating Implementation Process as well as Implementation Work Products

It is important for an evaluation team to assess the process of producing an implementation in addition to evaluating the work products of the implementation process. This is because the activities within the implementation phase can produce an overwhelming amount of material, in fact, more material than the design phase. For example, the work products include the source code, software development folders (SDFs), test reports and user documentation. The evaluators need to focus on the work product. Yet, to gain confidence that the implementation contains few remaining flaws, given this amount of information, the evaluators need to assess the implementation processes. This is in line with the idea that one can not have a truly good product without a solid process in place, which is the lesson being learned more and more in development programs.

The software development plan (SDP) needs to be updated for the implementation and integration phases. The SDP may not have been revisited since the project's start. Therefore, the update needs to reflect the additional information obtained during the previous phases. Aspects to be revisited include the development methods to address error handling, data base access methods, and service calls. It is also important to ensure that the test plans dovetail with the SDP. Since the maintainability of a system is directly related to the tracking and configuration management system maintained during development, it is important to ensure that all related plans are updated and tailored appropriately. If an error shows up, the responsible process can be revisited, (i.e., via unit testing, coding, etc.), and the necessary corrections made. In summary this identifies the need for process that is self-correcting. Consequently, when an evaluation team sees that the processes are in place and are being used, then there is increased confidence, and thereby increased assurance, that, as stated in the TCSEC, "no design flaws and no more than a few correctable implementation flaws. . ." are found "with reasonable confidence that few remain."

3 WHAT MAKES A GOOD IMPLEMENTATION?

This section defines those attributes of an implementation that make it "good." This section also provides guidelines to assist the evaluator in assessing the quality of an implementation. According to Beizer [2],

> *Good software works, meets requirements, is robust, and is easy to understand, easy to integrate, and easy to test.*

We fully endorse this brief definition and elaborate it in this section within two categories of "goodness," *viz*.

- the faithfulness of an implementation to the design that it is based on ("works and meets requirements"); and
- the intrinsic quality of an implementation ("is robust, and is easy to understand, easy to integrate, and easy to test").

The following two subsections deal with these two categories and close with comments on how the evaluator is to assess the implementation, based on the category of implementation quality given in the respective subsection.

It is important to be aware that a final evaluation of the implementation cannot be made until the software integration phase is nearly complete. Earlier preliminary evaluations can be made of the code, but until the end of software integration, the code is likely to be too volatile for any evaluation to be considered final. The risk to successfully completing an evaluation that is associated with deficiencies discovered in the implementation covers a wide range, depending on the specific deficiencies. A few localized deficiencies discovered in the code represent a low risk. If, however, a code deficiency is pervasive (as in the case of poor style throughout), a high risk results. There is also a high risk if major functionality is missing. If software documentation is missing or substandard, but the code is good and is based on a good design, the risk is probably low.

3.1 Correspondence to the Design

Correspondence of the implementation to the design is the basis for a significant part of the implementation evaluation. This is because guideline and criteria documents (notably the TCSEC) require a demonstration that the implementation corresponds to the design. Such a demonstration provides assurance that the implementation satisfies the security policy, because it is also a requirement that the design be shown to satisfy the security policy. This is a sensible approach, since the abstraction of the system given by the design is intermediate to the abstractions of the security policy and the implementation.

Within a system development, correspondence of the implementation to the design provides assurance in three areas. These areas are:

- understandability,
- predictability, and
- presence of protection mechanisms.

The first area is *understandability*. At the implementation phase, the design is presumed to be understandable, and by tracking to the design, the understandability of the implementation is enhanced.

The second area is *predictability*. One kind of security risk in a system is the potential for functionality in the implementation that is inconsistent with that in the design. This might include extra functionality hidden somewhere, or missing or modified functionality.

In any of these cases, the result is actions that the design does not anticipate. This concept of predictable functionality is the basis for acceptable behavior of the software.

The third area of assurance is the *presence of protection mechanisms*. This extends beyond simple predictability. It underscores that the protection mechanisms determined to be necessary in the design phase actually exist in the implementation.

The the following paragraphs present four evaluation procedures to be used in determining whether these three assurance areas are met. The four evaluation procedures are:

- processing structure correspondence,
- object definition and relationship correspondence,
- interface correspondence, and
- processing detail correspondence.

The following sections explain each evaluation procedure in terms of its contribution to assurance and in terms of its performance.

3.1.1 Processing Structure Correspondence

Processing structure correspondence consists of comparing the structural elements and their relationships in the design with those of the implementation. Considerable variation exists among types of representation in both areas. In the design, both graphical forms (structure charts and Buhr diagrams are examples) and textual forms (Ada skeletons and pseudocodes, for example) exist. In the implementation, a primitive division of the code by file may be used, or sophisticated language features may define structural elements (e.g., Ada's packages, Modula's modules). At a lower level, functions, procedures, and tasks are used.

Beyond breaking the code into manageable segments, the structure usually has the purpose of imputing certain semantics to portions of the processing. For example, the parameters of a function define the interface between the function and its caller. Also, an Ada package may be used for abstraction and data hiding (as with the definition of abstract data types and supporting methods).

In order to be able to determine whether the implementation corresponds to the design, compatibility must first be shown between the form of the design representation and the implementation language used, including how the language is used. An effective evaluation of the implementation should consider these areas of compatibility, as well as an item-by-item comparison. The result of the latter should be a close relationship between design elements and implementation elements.

3.1.2 Object Definition and Relationship Correspondence

The data representation of the design should be compatible with that of the implementation. One would expect an object-oriented design to be realized by an implementation that is equally object oriented, possessing, for example, abstract data types. It should be possible to describe the object definitions and relationships of both the design and the implementation in terms of abstract data models, and the two data models should in fact be identical. Ideally, such an abstract data model should be presented in some form in the design.

Not only should classes of and relationships among data entities exactly map between the design and the implementation, but the instances of the entities should be consistent. This includes the variables and constants of each scoping domain. Examples are variables and constants local to functions and those that are global with respect to individual larger

scopes (e.g., packages). More pervasive (i.e., whole-program) global instances are discouraged.

Note that in the case where an object-oriented design and programming approach has been followed, considerable duplication of code and data may exist. For example, there may be a "car" object and an "airplane" object. Each object may have a service function of "acceleration." The nature of the service is tied directly to the object with the consequent duplication. This type of duplication may be quite acceptable as long as the duplication (e.g., the service) is used by the implementation. The issue of duplication of code and data resulting from an implementation of an object-oriented design approach will need to be mutually agreed with the evaluation team and the development team. In particular, one needs to ensure that the object life-cycle history "states" cannot reveal any information that is incompatible with the design or with the security policy.

3.1.3 Interface Correspondence

The analysis of system interfaces completes the analysis of the system structure. The interfaces define the communication among structural elements. Interfaces are defined in terms of processing capabilities offered and information that may be transferred across them. That information is in turn described in terms of the data types that characterize it and the semantics of each interface as expressed using these data types. Interfaces must be defined in these terms in detail. Interfaces must be explicitly defined in these terms, both in the design and in the implementation, and, of course, these definitions must be consistent.

3.1.4 Processing Detail Correspondence

The assurance area of *predictability*, as indicated in Subsection 3.1, addresses the risk of either extra or missing functionality in the implementation, relative to the design. This is the fundamental issue within correspondence of the implementation to the design. This subsection assumes that correspondence analyses have been performed for processing structure, the data descriptions, and the interfaces, as in the preceding subsections (3.1.1, 3.1.2, and 3.1.3). Those analyses having been performed, the functionality may be analyzed by examining each structural element separately in terms of the processing it is to provide. This examination can be limited to those elements that actually define processing such as functions, procedures, and tasks. These are expected to be closely related between design and implementation. The analysis consists of comparing these relationships to assure identical functionality.

Besides assuring that implementation functionality derives directly from the functionality specified in the design, it is important to focus especially on the functional areas that implement the protection mechanisms in the system. The areas of protection mechanisms, as indicated in the TCSEC (for Class B3), are:

- discretionary access control (paragraph 3.3.1.1, Discretionary Access Control),
- object reuse (paragraph 3.3.1.2, Object Reuse),
- labels (paragraph 3.3.1.3, Labels, and subparagraphs),
- mandatory access control (paragraph 3.3.1.4, Mandatory Access Control),
- identification and authentication (paragraph 3.3.2.1, Identification and Authentication, and subparagraph),
- audit (paragraph 3.3.2.2, Audit),
- system integrity (paragraph 3.3.3.1.2, System Integrity),
- separate operator, system administrator, and security administrator functions (paragraph 3.3.3.1.4, Trusted Facility Management), and

• trusted recovery (paragraph 3.3.3.1.5, Trusted Recovery).

Note that these portions of the TCSEC are specifically selected because they are the paragraphs that deal with system functionality. The concept here is that the implementation is to be inspected to assure that these functional areas from the design are faithfully represented in the implementation. The implementation is to be further inspected to assure that the representation is not only faithful, but also appropriate within the implementation. For example, the implementation may reveal potential for object reuse problems that are not visible in the detailed design. All such problems must be dealt with in the implementation because the design may not deal with all of them explicitly.

This is because the level of detail of the implementation is even greater than that of the detailed design. These implementation flaws can only be discovered by examination of the code. The suggested approach is a full survey-level review, as well as in-depth review of selected portions. This is explained further in Section 3.2.3.2.

3.1.5 What to Look For

Subsection 3.1 has described four procedures that work together to determine the correspondence of the implementation to the design. By their nature, the procedures must be performed in the order given. The following statements summarize them.

1. Check that the processing structure given in the design (in terms of such documentation as structure charts or pseudocodes) is realized in the implementation. Fundamental to this is a comparison between the indicated processing divisions (units, routines, etc.) in the design and their analogs (packages, procedures, functions, etc.) in the implementation. Consistent relationships must exist between the design elements and the implementation elements.
2. Check that object classes, objects, and their relationships are exactly related between the design and the implementation.
3. Check that interfaces of equivalent structural entities are themselves equivalent.
4. Check that the processing, or functional, details of equivalent processing entities are in turn equivalent.
5. Given that the design has successfully passed evaluation, significant encapsulation will be embodied in the design. It is critical that this encapsulation be carried through into the implementation, using whatever language features are available.

Because an evaluator has the need for information to perform these checks, it is necessary to have complete and timely design documentation. This implies a well-defined software development process (such as DOD-STD-2167A) that is executed in a reasonable way. By this we mean a way that is sufficiently disciplined to gain maximum vlaue from the process while preserving enough flexibility that necessary corrections to earlier results are always possible. An assumption is made in this subsection that software baselines are maintained in a program library under strict configuration control, and the current design baseline has been accepted previously by a design evaluation team.

3.2 Intrinsic Quality of the Implementation

This subsection deals with characteristics of the implementation that are not directly related to the design. That is, an implementation might capture the design and yet be unacceptable for some of the reasons given below. These concepts are well developed in a number of standards. A number of developers have their own guidelines. Most

government contracts involving software development require guidelines, often included within the Software Development Plan (SDP). As an example, the Ada Productivity Consortium has produced Ada usage guidelines [1] that are in general use. These Guidelines are tailored to specific projects.

3.2.1 Language-Independent Quality Characteristics

It is important that code be readable so that it can be easily understood and maintained. Understandability is necessary for reviewability, in order to be sure that not only does the implementation correspond to the design (Subsection 3.1), but also that nothing extra has been intentionally or unintentionally included (examples: temporary debugging code and "Trojan horse" code).

For example, an Ada program would be unreadable if a non-standard style were used such as unusual formatting, or if it contained many small packages. On the other hand, an Ada program might appear to be readable but contain such usage of overloading and generics that its functionality is much more complex than it would immediately appear; the appearance might in fact belie the actual functionality.

Specific readability characteristics include naming conventions, commenting practices, and textual formatting characteristics, such as indentation and spacing. An important characteristic is *consistent* coding style. Multiple styles cause difficulty in reading an implementation, even if each is itself a readable style. This point supports the importance of definition and enforcement of coding standards.

Also, consider possible weaknesses of the particular implementation language that might allow it to be abused to hide extra code. For example, a weakly typed language that allowed an array name to be used as a procedure name would provide the dangerous facility of allowing pre-compiling code, manually placing the object code in an integer array and executing the array.

To enhance the readability, understandability, and reviewability, implemented code needs to meet accepted coding standards and conventions. These standards and conventions should be part of the Software Development Plan or Software Standards Document, or be part of company standards referenced by those documents. It should be noted that flaws in coding style result not merely from missing or ignored coding guidelines. In many cases, coding guidelines (government or contractor) are out-of-date or simply ill conceived, and so mandate poor practices. One example is a statement in an existing government contractor's coding guideline that insists that every line of every program be commented. Such a practice often results in redundant comments and usually detracts from the readability and understandability of the code.

The remainder of this subsection describes specific coding standard details.

Program logic should be easy to follow. For example, decision-action pairs should be physically close together in the code. Likewise, initialization of variables should be close to their first use. Expressions should not be too complex. Often careful formatting can make an expression easier to follow. Case statements should always have a default action. In a language like C with "drop-through" cases, an explicit comment should be used to indicate where a drop-through is intended (i.e., where no *break* at the end of the case is desired).

Comments should be used in such a way as to increase clarity. Too many comments (or too much detail in comments) detract from the readability of the code. Comments should be used to address the purpose or scope of the code, explain the algorithm used, or note any assumptions or limitations in the code. Comments should not duplicate anything in the code (e.g., identifying packages imported via `with` or `include` or restating the code). In particular, commenting every line in code usually reduces understandability. Also, one

should be aware that comments can be used maliciously to distract from the true intent of the code.

Layout standards are critical for readability. These include indenting, use of blank lines or page breaks for separation between procedures, and placement of element (package, procedure, etc.) header comments.

Naming conventions need to be consistent for types, variables, constants, procedures, functions, packages, and other language constructs, including the use of underscores, special characters, abbreviations, etc. Proper naming can contribute to the readability of the code. In general, the full name space of the language should be used to provide readable code. This may involve updating older coding standards developed for languages that restricted the length of names.

Optional syntax usage should be standardized. For example, most organizations require the use of named parameters in procedure calls in Ada, that is:

```
Get_Buffer_Information (
      Input_Stream      => Telex_Data;
      Buffer            => Ethernet_Packet;
      Buffer_Filled     => FULL_PACKET;
      Overflow          => MORE_DATA_IN_NEW_PACKET;
      New_Buffer        => Overflow_Ethernet_Packet);
```

instead of

```
Get_Buffer_Information (Telex_Data, Ethernet_Packet,
                  FULL_PACKET, MORE_DATA_IN_NEW_PACKET,
                  Overflow_Ethernet_Packet);
```

This often makes the code more readable, and it certainly makes the code more robust. Another common standard for optional syntax use is to require names to be fully qualified (e.g., `Package_Name.Procedure_Name` or `Package_Name.Data_Structure_Name`).

Size and complexity, including nesting and depth of nesting of control structures (e.g., if-else), procedures, generics, and custom overloading are important areas for standards. In this category, improper usage can lead not only to a lack of readability, but also to confusion, giving an appearance of readability that is not valid.

Exception, interrupt, and error standards are likely to need refinement during the coding phase to provide a common way of coding these items that is in line with the specific software and hardware architecture.

Return status checking from data base calls and system calls requires standards that are likely to need refinement during the coding phase to deal with the specific target environment and product architecture.

3.2.2 Use of Implementation Language

An element of security evaluation unique to the implementation phase is assessment of the use of the implementation language features. Subsection 3.2.1 described the use of features in code in general. This subsection points out the need to apply those concepts to specific language constructs. It is assumed that the chosen implementation language has a well-defined and well-documented language model in terms of both data and computation. These define how the programmer fundamentally expresses himself in the language. In order to obtain the greatest benefit from the language model, it is important that the programmer use the language in the way it is meant to be used. If, for example, an Ada program makes much use of global variables, then the programmer has worked at cross

purposes with respect to the language model. Instead, the programmer should use packages and processes in a way that maximizes data hiding and functional layering.

Most modern languages have features that can be used to increase readibility, ease of change, and robustness of the code. For example, by using the Ada attributes *'FIRST*, *'LAST*, and *'RANGE*, the change of a range can be done in a single place, i.e., where it is declared. Other examples are the use of named constants for literal, enumerated types instead of small integers, dynamic allocation of arrays, and definition of abstract data types.

The use of multiple languages in software implementation requires special consideration. In the case of Ada and an assembly language, for example, facilities for integrating assembly code into the Ada environment are well defined. These facilities may be provided by the Ada tool kit, resulting in little confusion and therefore low risk. On the other hand, two languages (such as Ada and Cobol) that both need to "control the universe" can create a problem. This occurs when, for example, both language environments expect to control the name space or to specify the initial execution address. Mysterious code that might be written to get around such a problem represents a significant security risk, regardless of how well documented. Program requirements that necessitate such work-arounds increase security risk and as a consequence, increase the risk of an unsuccessful attempt at system certification.

3.2.3 What to Look For

Following is specific guidance to the evaluator in examining code for implementation-specific characteristics. This includes a description of specific implementation products and specific characteristics to look for.

3.2.3.1 Where to Look

The implementation work products are composed of more formal and less formal products. The more formal are those that are typically deliverable items, including code, documentation, and test documents. Documentation is of several forms, including user documentation, documentation for operations, design documentation, and maintenance documentation. Only the user and maintenance documentation are specifically associated with the implementation phase of development. The test documents include test plans, test procedures (including test cases), and test results. A key element in product quality (with particular application to trusted systems) is consistency. This includes self-consistency of each product and consistency among products. This consistency characteristic is an important area of analysis.

The less formal products consist of development records, which ought also to be available to the evaluators (cf. 5.1 and 6.1). They are typically in the form of a Software Development File (SDF), Software Development Notebook, Unit Development Folder, or Programmer's Notebook. These terms are essentially synonyms and indicate a collection of material pertinent to the development of a software module or unit. Contents typically include the requirements, design, technical reports, code, test plans or test cases, test results, problem reports, schedules, and notes for the module or unit. Also included are unit-specific management indicators, including tracking information; and Quality Assurance (QA) and Configuration Management (CM) information such as discrepancy reports (DRs) and their resolutions. Multiple versions of these items are kept as development is under way, and not merely as final versions. It is preferable to have the design Program Design Language (PDL) and the code in a CM library such as Source Code Control System (SCCS) or Code Management System (CMS) rather than as hard copy in the notebook. The use of CM on-line tools can provide histories and differences between versions as well as ensuring that the proper version is available for examination. Similarly, much of the SDF may be kept on-line in the development environment instead of in hard copy form. The milestones associated with the module (e.g., code walkthroughs or code inspections,

hand-over to integration testing, etc.) can be kept as part of the SDF or as part of a management tracking system. Whatever methods are chosen should be in-line with the project plans and should readily provide information on the success or failure of these milestones to the evaluator.

3.2.3.2 Specifics of Evaluation

All the standards relating to clarity of the code given in Subsection 3.2 should be applied, providing an objective measure of clarity. Nevertheless, clarity of the code may still be difficult to quantify. A complex algorithm may require complex code that is difficult to understand but is not overly complex. The evaluator might do well to consider the level of frustration that results from the learning process rather than difficulty of learning the details of a piece of code. A high level of frustration may indicate overly complex code, inconsistent style, inappropriate language use, unavailable documentation, etc.

The evaluator needs to focus on consistency of the code as well as clarity. Different portions of the code may seem to meet the clarity criterion, but because they are inconsistent in usage may result in a misleading understanding of the code.

In summary, the evaluator needs to have confidence that the implementation language has been used appropriately and that the code does what it is supposed to, and nothing else.

In a large system, it is unlikely that a team of evaluators can review in full detail every line of code. Thus, a useful approach may be, after a complete understanding of the architecture, to perform a survey-depth review of all the code, and then spot-check in detail specific portions of code. Subsection 5.3.2 provides guidance for selecting portions for detailed review, based on what is likely to be especially at risk.

4 WHAT MAKES A GOOD IMPLEMENTATION PROCESS?

This section considers the question of what makes a good implementation process for trusted system development. The examination of the implementation process is important to the evaluators because the evaluators' confidence in the process provides more confidence about the implementation than can be gleaned from just an examination of the work products themselves. The following eight sub processes to the implementation process are important to the evaluator of B3 systems:

- Project Control,
- Coding,
- Unit Testing,
- Integration,
- Integration Testing,
- Documentation,
- Support Tool Development and Use, and
- Configuration Management.

Subsequent subsections of this section explain concepts relevant to each of these processes and give suggestions on what evaluators should look for in these processes.

4.1 Project Control Process

The main purposes of the project control process or function are to plan the implementation and integration activities, allocate the appropriate resources to the activities, and monitor the activities and processes. Reviews and inspections of the implementation and integration products and the resulting rework need to be included as activities.

The project control process requires visibility (metrics) and outputs this information as management indicators [4], [12], [17], [29], and [31]. The accepted SDP should have specified the management indicators to be used for the implementation phase. Many items can be monitored in a project. These items can be grouped as the "five P's":

- Plans — providing visibility of the target or goals,
- Progress — providing visibility as to what has been accomplished and at what cost,
- Product — providing visibility of the quality of the creations of the project,
- Process — providing visibility of the effectiveness of the methods, procedures, and tools used on the project, and
- Peripherals — providing visibility of the effectiveness of the project support activities (e.g., training, administration, etc.).

The focus for the evaluator is the indicators on Product and Process. The Progress and Peripheral indicators, other than completion indicators, are of less concern to the evaluator. These indicators will form a substantial part of the material for evaluating the implementation. The evaluator needs to consider the reasonableness, the effectiveness, and the timeliness of the indicators when evaluating the project control process. The evaluator should remember that anything not measured is unknown. Because it is not cost effective to measure everything, particularly to a very detailed level, metrics should be used to measure the most important aspects of the project. From a security evaluation point of view, the project aspects that must be measured include:

- the enforcement of project standards,

- the enforcement of the change control process,
- the completion of walkthroughs and inspections, and
- the closure of rework items.

The collection and analysis of these management indicators ensure that the development process was a controlled process. This is assessed by verifying that the procedures identified in the software development plan were followed, that the change control process was followed, that the reviews and testing occurred, and that the resulting rework was monitored and completed. These procedures and reviews should have prevented defects and any extraneous insertions of code or functions into the product.

The software development plan should have specified the review types and frequencies per unit of the product. In general, there should be a code walkthrough at an early stage of the implementation and a code inspection following unit testing for each unit. The unit test case and test results should also generally be inspected before integration. The software development plan should also have specified the method for defect resolution or rework and how this would be tracked to completion. The project control process should keep the dates of the walkthroughs and inspections, the defect counts found in inspections, the dates each unit was available for integration, the defects found by integration testing, and the dates the units passed integration testing. There should be some cluster or Pareto analysis of the causes of defects. Typically, the responsibility for performing these analysis is assigned to the technical lead or to the software quality organization. Ideally, the project control process uses these analyses to improve the implementation process during the implementation phase.

The SDP should have specified how the units would be determined, but in general, the units should have been chosen such that each unit can be inspected and tested independently. The units should be sized such that an inspections would not exceed two hours. This typically means a maximum of about 50 pages of text (as in test cases) or a maximum of about 25 pages (or fewer) of source code per unit.

Because evaluators and project managers need to be familiar with the different types of reviews and their appropriate uses, the following information is provided on technical review, walkthroughs, and inspections. Further information is available in [8], [10], [11], and [16].

4.1.1 Technical Reviews

Technical Reviews evaluate software products for defects and for conformance to specifications and plans. Typically two to three people, including the technical leads, perform Technical Reviews. The developer of the software product(s) presents a relatively large amount of material (>100 pages) to the reviewers. This type of review is usually not sufficient to ensure that the product is correct and complete enough to be input to the next development life-cycle phase.

4.1.2 Walkthroughs

Walkthroughs are a type of peer review to

- detect defects,
- examine alternatives, and
- provide a forum for communication and learning.

Walkthroughs typically include two to seven participants and may include technical leads as well as peers. Typically the developer of the software product leads the walkthrough and presents a relatively samll amount of material to the walkthrough participants.

Resolution of defects or issues brought up in the walkthrough is made by the developer. Verification of the resolution and changes is left to other project controls and is not a function of the walkthrough process.

Walkthroughs are better than other review techniques for exploring alternatives and thus are most applicable for reviews of products at early points in the particular development life-cycle phase. Thus it is appropriate for the implementation process to use walkthroughs for examining implementation or code modules before completion, or for examining the first modules produced in the implementation phase. Walkthroughs may be subject to domination or intimidation, particularly by the technical leadership or by the developer of the software product being examined.

4.1.3 Inspections

The primary objective of software inspections is to detect and identify defects in software products (e.g., source code, unit test cases, etc.). Inspection is a rigorous and formal type of peer review that:

- verifies that the software product(s) satisfies specifications,
- verifies that the software product(s) conforms to applicable standards,
- identifies deviation from standards and specifications,
- collects defect and effort data, and
- does not examine alternatives or stylistic issues.

Inspections may be applied to any written document but are especially applicable to those products that are to be input to the next life-cycle development phase. The goal of inspections is to never pass an incomplete or incorrect product to the next phase of the development life cycle and thus to resolve the defects as early as possible. Michael E. Fagan of IBM further developed inspections into Fagan's Inspections in the 1970's to also improve the software engineering process [10] and [11].

Inspections are conducted by peers of the author or developer and typically comprise three to six participants called inspectors. The process is led by a moderator who is impartial to the software element being examined. The inspectors identify and classify defects including ambiguous, inconsistent, missing, extra, or wrong items. Inspectors are often assigned specific roles for the inspections such as to verify the conformance to standards, to represent the "customer" of the software product (e.g., the integration testers), or to perform the data flow analysis or a data base analysis, etc. Defect resolution is mandatory and rework is formally verified by the moderator.

In addition to the specific roles assigned to the inspectors, the following special roles are defined for the inspection process:

- Moderator — The moderator is the chief planner and meeting manager for the inspection meeting. The moderator establishes the preparedness of the inspectors and issues the inspection report.
- Reader — The reader leads the inspection team through the software element(s) in a comprehensive and logical manner.
- Recorder — The recorder documents the defects identified in the inspection meeting and records inspection data required for process analysis.
- Inspector — The inspector identifies and describes defects in the software element(s). Inspectors must be knowledgeable in the inspection process and in the viewpoint they represent. Inspectors also have checklists to guide them

toward finding as many common defects as possible. (*See The Art of Software Testing* [22] for a language independent checklist for code inspections.)

- Author — The author is responsible for ensuring that the software element meets its inspection entry requirements. The author contributes to the inspection based on his special understanding of the software element. The author is also responsible for performing any rework required as a result of the inspection. The author is not allowed to perform any other role (i.e., moderator, reader, or recorder).

Inspection meetings are limited to two hours and follow the following agenda:

- Moderator introduces the meeting.
- Moderator establishes the preparedness (e.g., time spent, defect count, completion status) of the inspectors and reschedules the meeting if the inspectors are not adequately prepared.
- Reader presents the material to be inspected.
- Inspectors identify the defects and defect categories.
- Recorder records the location, description, and classification of each defect on the defect list.
- At the end of the meeting, the recorder reviews the defect list with the inspectors.
- The author may schedule a "third hour" to discuss alternative solutions to reported defects.
- Inspectors bring the inspection meeting to an unambiguous closure by deciding on the disposition of the software element. This disposition may be:
  - –Accept - The software element is accepted "as is" or with only minor rework that requires no further verification by the inspection team.
  - –Verify Rework - The software element is to be accepted after the moderator verifies rework.
  - –Re-inspect - Schedule a re-inspection to inspect the revised product. At a minimum this re-inspection examines the product areas modified to resolve defects identified in the last inspection. Re-inspection is usually necessary for defects classified as major defects.

Inspections and walkthroughs have been found to be effective in finding 30% to 80% of the logic design and coding errors in typical programs. Inspections typically achieve closer to 80% of the errors detected by the end of testing. Part of this success is because an inspection or walkthrough essentially uses people other than the developer to "test" the software element. These techniques are also successful in lowering the error correction costs because they typically locate the precise error when they identify a defect. These techniques also expose a batch of defects at one time, thus allowing the defects to be corrected en mass. On the other hand, testing normally exposes only the symptom of the defect, and then the defects are located and corrected one by one. At least one study has shown that the inspection process tends to be more effective than testing at finding certain kinds of errors (e.g., logic errors) while the opposite is true for other types of errors (e.g., errors in requirements analysis, performance problems, etc.) [21]. Thus both inspections and testing are necessary to ensure defect detection and correction.

4.1.4 Encouragement for Improved Standards

This chapter expects that standards will be in place to which developers are expected to adhere. Such standards encourage a consistency of product that makes review more effective and usually upholds a standard of quality consistent with that required for B3 systems. A note of warning, however, is necessary regarding development standards. Some standards contain requirements that run counter to modern accepted practices and to the guidelines put forth in this chapter. Often these standards were developed and accepted to work around limitations of languages, compilers, and other tools. These limitations required extreme practices to create a product that was understandable. In addition, some problems with standards are a result of a lack of understanding of general goals or of a view of programming at considerable variance with today's views.

To emphasize these remarks, we provide some examples. A prominent aerospace contractor at one time enforced coding standards that prohibited normal use of Ada tasks. The specific statement (apparently oriented toward assembly language programming) was that routines are prohibited from having multiple entry points, thus disallowing the Ada rendezvous construct. Another of that contractor's standards required a comment on every line of code. That may have been appropriate for assembly language (though even that is not clear) but would only be obstructive for a higher-order language. The same standard made no mention of structured comments. Another example is that the standard required a three-letter code for the module name be part of the name of every entity *within* the module. This is certainly unnecessary for languages with separate name spaces for different modules, and adherence to such a standard considerably reduces readability of the code. (Note that the objection is to a standard requiring a duplication within a module or package and not an objection to standards that require an identification of the subsystem or CSCI to which the module belongs. The latter standard benefits the maintenance of the software system.)

The point of these observations is that this guideline encourages the use of standards that not only enforce consistency but also support understandability and reviewability. The existence of obsolete standards with obstructive requirements is not to be considered a legitimate excuse for failing to meet the standards of understandability and readability contained in this chapter. Contractors who demand that their programmers adhere to such obsolete standards should expect significant difficulty in achieving positive certifications of their implementation products and processes.

4.1.5 What to Look For

Evaluators should look at the inspection records for the source code and unit tests. The preparation times for the inspectors and the types and frequencies of defects detected should indicate that the inspections were thorough. The follow-up should indicate the correction or closure of all items identified in the inspection defect list. Re-inspection should have occurred for any units which had major defects indicated in the original inspection. (Typically only 1 or 2 major defects per 50 pages is allowed without triggering re-inspection, but the SDP may have specified a different trigger or different definition of major defect.)

Evaluators should see that the implementation was compared to the design for each of the units (i.e., the "higher level" specifying document is part of the entry criteria for the inspection). Comparison for faithful realization of the design and for detection of missing, added, or incorrect functionality should be part of the inspection process for the code and unit tests.

Evaluators should see that the project standards were enforced either by automated detection of deviations or by inspection of the implementation work product(s).

Evaluators should also look at the progress metrics. Specifically, they should look at the completion dates and final inspection dates for the units. Ideally there should not be large deviations between the planned completion dates and the actual completion dates — although replanning is common among software development projects. Occasionally, a large number of units are all completed at the end rather than in a more leveled out order. This *may* be an indicator of problems with the implementation because it may mean a slighting of the inspection process (due to overloading of the critical reviewers) or a rushing of the completion to meet an arbitrary deadline. In these cases, the evaluators should examine these "late" units and their the inspection and testing records to ensure that the quality of the units is adequate (e.g., the units really are finished).

Evaluators should also look at other progress metrics such as manning curves by experience levels and arrival of peripheral and support tools. These progress metrics can indicate significant deviations from the approved plan and, therefore, indicate that the evaluator should analyze the recovery plans or adjustment made to be comfortable that the implementation quality did not suffer.

4.2 Coding Process

The coding process produces the source code. Implementors code the modules according to the design specifications to produce a faithful representation of the design. An object-oriented design naturally leads to object-oriented programming, but the coding process is the same whether object-oriented or structured techniques are utilized. The coding process for a module ends when the module has not only passed its unit test case but has also been successfully integrated and tested as part of the integrated system.

The coding process can generate change proposals. Actual changes and proposed changes are considered part of the configuration management process because the changes must be managed so that the code can be a faithful representation of the design.

Coding may proceed in a top-down, bottom-up, or combination of these two methods. It is very common to require many of the low level modules that form a support environment or project library to be completed and tested early.

The source code that the implementors write needs to adhere to the "rules" of the project coding standards. There may be some exceptions to the project standard "guidelines," but these exceptions should be rare. Those exceptions must be documented in the source code and in the SDF for the module.

To the project standards defined near the beginning of the project which are largely based upon company standards prior to the project, the implementation phase should add detailed standards based upon the coding experience to fit the system being developed. The purpose of detailed standards is to produce a consistency in the code to improve the readability and understandability of the code and to improve the reliability of the code. Typically these are standard ways of coding items such as checking the return status from data base calls and system calls or dealing with exceptions, interrupts, or errors. These common ways of coding need to be thoroughly tested on the target platform before incorporation into the code. The evaluators should see evidence of this consistency in the code and evidence that the common methods were adequately tested.

Adherence to standards should be part of the code inspection checklists. Automated tools can assist this inspection process by providing evidence of adherence or by enforcing adherence. Certainly, many modern compilers can assist by providing evidence of a clean compile and by providing warning messages about nonstandard constructs. In addition there are products such as *lint* (for C programs) or *Adamat* (for Ada programs) that can provide additional warnings. Some of these products can even be tailored somewhat to support the project standards. *Pretty-printers* can be used to ensure consistency with the

standards for indenting, providing blank line separation between functional units, and other layout details. Automated tools can check the size and calculate a complexity measure for the procedures, functions, and tasks in the unit to provide evidence that the size and complexity standards have been met (Section 5.3.2.1 gives more details on a few of these measures.). Evaluators should understand the project use of these tools and the limitations of the tools so they can assess the contribution of these automated tools to the overall adherence to coding standards.

Part of the coding process is maintaining the SDF for the module(s) being coded. Of particular importance is the maintenance of the status of the modules. This includes, for example, resolution of issues or rework due to walkthroughs and inspections, the identification of the most current version being developed prior to turning the source over to CM, and the discrepancy report (DR) or change proposals on the module that are in progress. The SDF should also contain any "historical" information about the unit such as memos, minutes or action items from technical interchange meetings, etc., that might be helpful to understanding the development of the unit.

The coding process may introduce special features into the code to assist testing and debugging of the code. These features need to be removed before system testing once they become extraneous. The coding process may also take advantage of temporary modifications such as separate compilation features in Ada (where even smaller units than are in the design are separately compilable). Again, these temporary changes should be removed when no longer required and before system testing begins. These changes also often require modifying the compilation/link lists or *make* files for CM.

Evaluators should see that the coding process yields consistent code that meets standards not only for implementation style but also for error checking and handling. Evaluators should understand any tools used to generate code or to check code (e.g., for standards) and should also understand the limitations of those tools. It should be apparent that any extra code for debugging has been removed as part of the coding process. Evaluators should also see that information on the lower level software, hardware, and recommended language usages was readily available to the coders. The ideal coding process should be self-correcting — that is, lessons learned from inspections, unit testing, and integration testing of the early units should be distributed to the implementation team to improve the other units.

4.3 Unit Testing Process

The unit testing process usually goes along with the coding process and may be a tight loop. Unit testing is the process of doing detailed, "white box" or logic driven testing on the modules composing the unit in addition to "black box" testing. In other words, the unit tester generates test cases based not only upon the design specification or parameter interface (the "black box" case) but also based upon the algorithm or logic (the "white box" test cases).

In general, the "white box" testing should ensure that close to 100% of the source code statements are covered by the testing. Exceptions to coverage should be examined carefully to ensure that these lines do not represent "dead code" or code which should be covered by another test case. The common code added as part of the standards does not have to be covered in each incorporating unit but should be thoroughly tested before incorporation. Unfortunately, even 100% coverage of the source is not sufficient testing as every statement could have been executed at least once, yet many paths through that source may not have been tested at all. Testing every path in even relatively simple procedures can be infeasible because of the combination effects (particularly if there are nested *if*s within loops). Therefore a stronger path coverage criterion is that there should be enough test cases to cause each condition in each decision statement to be executed at least once; each

decision outcome to be executed at least once; and each point of entry to the procedure, function, or task be executed at least once. Combinations of these condition outcomes should also be tested by the test cases.

In general the "black-box" testing should consider equivalence partitions and boundary values tests. Equivalence classes are values such that any errors or lack of errors found by that value are representative of the results from any other member of the equivalence partition. If an input condition specifies a contiguous range of valid values, then the valid equivalence partition consists of any member of that range. The invalid equivalence partitions are the other possible values. Of particular importance in the invalid condition partition are the values at the boundaries of the range. Most programming errors occur at the boundaries of the valid ranges. Consequently, the use of boundary value tests or specifying inputs just outside the boundary and just inside the boundary form a very necessary part of unit tests.

In addition to the above unit testing designed to help ensure that the unit does what it is supposed to do, unit testing of trusted software also needs to help ensure that nothing that was not supposed to happen doesn't happen. This is a difficult, often infeasible, task to accomplish in general. It is important, however, that unit test cases specify the exact expected outcome and that the results are inspected to ensure that nothing was added to that outcome. This much can be done.

Whenever possible, unit test cases should be scripted or otherwise fashioned so that regression testing can be automated. When defects are found during the integration and system testing, the unit test cases need to be rerun on the fixed units. Without automation of the regression testing, many defects introduced by the correction of other defects can slip through since ad hoc testing by the fixer is rarely systematic or complete enough.

It is often found that unit testing is harder than it seems. The unit tester must have the philosophy of attempting to break the code instead of the intent of showing that it works. Thus it is often beneficial to have a separate, knowledgeable person other than the coder develop and execute the test cases. In fact, it is sometimes helpful to have testing specialists do the unit testing to provide the required skepticism and "diabolicalness" for adequately testing the units.

Unit testing for object-oriented designs and object-oriented programming is likely to require a test driver or harness. The unit testing of the services of the object is equivalent to the previously described unit testing. In addition to this testing, the unit testing for object-oriented "packages" needs to ensure coverage of the life cycle of the objects. That is, object-oriented techniques require unit tests of the states and state transitions of the objects.

The use of Ada generics packages and similar techniques, while often desirable from a coding and design point of view, can complicate unit testing. Coverage analyzers may have difficulty with generic packages; so this part of the unit testing may require manual checking. Care must be used to unit test all equivalence partitions of the generic by testing the appropriate instantiations. The generic packages may play a very large part in the code if they have many instantiations. This increases the importance of testing the path combinations in the generic package code.

Evaluators should see evidence in the unit test records that every unit was tested. It should be clear to the evaluators that for every input test case the exact outcome was specified and obtained. All of the equivalence partitions and boundary values should have been covered in the unit testing. The unit testing process should have been thorough and methodical for all of the units. In general, units with a critical or high-use role in the trusted system should have had extensive unit tests to cover each decision statement and each of the possible outcomes from the decison. The test cases should have been inspected by personnel knowledgeable about testing and the software environment as part of the unit

testing process. Evaluators will have to use their own experience to judge whether the number of test cases and the orthogonality of the test cases was likely to have adequately tested the logic paths in the unit(s).

The unit testing process must include regression testing — unit testing needs to have been repeated for corrected or modified units. Sometimes there is a tendancy to *only* test that a bug was fixed and to skip thorough unit testing and testing to ensure that no new defects were injected by the fix. The regression testing is important for high assurance systems.

4.4 Integration Process

The Integration Process is the process of putting the units together to build capabilities. These capabilities are built specifically for integration testing. The software units that have successfully passed their unit test cases are placed under version control (i.e., CM control). The integration process then utilizes the CM managed versions to build the integrated components. The integration process also relies on having version controlled make files or compile/link lists. This version control is necessary to be able to provide version histories of units that have been integration tested. If there is no version control or if units can be copied into the integration test area by developers, it is impossible to know which code passed the integration tests. Version control also provides the ability to back-out changes that did not work. Evaluators should have confidence that the integration build process produces code from known sources in a repeatable process and releases the executable code to the integration environment. Evaluators should have confidence that this process can not be readily usurped.

"Early" integration builds may include some customization such as test harnesses or test drivers that are not part of the product or include some test hooks that will not be included in the final product. If the units contain test hooks or special code for testing, they will not have had their final code inspection. This is acceptable as long as later integration builds include only units that have had extraneous code removed and that have had their final code inspections.

The integration process is a precursor to the release process after testing. Therefore, much of the final CM release process can be worked out during the integration phase. The final CM release process, however, will have to include productization. One of the items to be worked out in the integration process is the "post-linking" of components. Just compiling and linking the separate programs (e.g., TCB, applications, etc.) is not enough to integrate them and allow them to communicate with each other or with the hardware. Environment variables or logicals, device names, directories, paths, and links have to be specified. The integration process must provide these other global elements in a controlled repeatable method.

Of special concern in the integration process is the integration of commercial off-the-shelf (COTS) products. These COTS products need to be included in the integration process. Many COTS products require the setting of special logicals or environmental variables and require the existence of special directories. Version histories of COTS utilized in producing integration capabilities and the elements that they depend upon need to be maintained.

4.4.1 Build and Customization Process

This area of assessment involves examining the procedure used for building an integrated version of the system. This includes the compilation, linking, and post-linking

processes, as well as customization for particular targets (either sites or special versions, for example) for testing. There may also be the necessity of integrating a COTS product.

4.4.2 Integration of COTS Products

In a system assessment, portions of the design and implementation specifically developed for the system can be examined for adherence to the given criteria. Most systems, however, contain products procured to satisfy particular component requirements. This is particularly true for hardware, but software products are also often procured "off the shelf." When this is the case for a system to be assessed with respect to a certain level of assurance, it must be shown that each COTS product appropriately supports the system assurance. This is also true for any software application hosted on a TCB; but with COTS there is no visibility into the source code.

This support for system assurance is provided in two areas. The first area is architectural coherence. This requires that a product fit appropriately within the system architecture as defined for the system's level of assurance. The second area is product security assurance. This requires that a product meet the level of assurance required for the product's context within the system. For example, if the product is part of the TCB, then its level of assurance would need to be the same as that for the TCB. On the other hand, if the product is fully constrained by the TCB and it performs no security-related function, then it would have no required level of assurance, for example, with any untrusted subject.

For the first area of support (architectural coherence), the product is merely being treated in the same way as any developed component of the system. That is, requirements are defined for the component that are assumed by other system components related to it. The requirements must, in turn, be met by the component. Some such requirements may be security requirements, i.e., requirements ultimately derived from the security policy of the system. If the relationship of the component to the system architecture is such that the component could be a security (or any other kind of) vulnerability, even while meeting its defined requirements, then it is incumbent on other portions of the system (i.e., portions of the TCB) to so constrain the component such that the vulnerability is annulled. Thus the thrust of the analysis in this area is to make sure that the set of requirements levied on the product component and on the remainder of the system is complementary such that security vulnerabilities are not introduced. A component is capable of causing a security vulnerability if, based on what the component is permitted to do (given the external constraints placed on it), it could cause a security policy violation. For example, consider a component whose job it is to transmit flow permission tables from an exterior interface to an interior portion of the TCB to be used there. That component is in a position to cause a security vulnerability and may do so, unless it has been verified not to corrupt the tables to an acceptable level of assurance, or else it is made incapable of corrupting the tables by applying a security lock to them.

The second area of support (product security assurance) necessitates the analysis of what nature of security requirements is expected of the product component. That is, it must be considered what level of security vulnerability would result were the component to violate its requirements. If the component is a portion of the TCB and is being depended upon to implement certain security controls then its level of assurance will probably be equivalent to the highest level required within the system. If, on the other hand, the component is fully isolated by the TCB and can in no way cause a security compromise, then no special assurance level is required of it. Intermediate levels of assurance may be necessitated if the component implements no security controls, but could be capable of a security compromise. Even the situation of having a seemingly small leak from CONFIDENTIAL to UNCLASSIFIED requires further analysis because the component may be part of a larger problem. The larger problem would occur if there is a path from very sensitive information (e.g. TOP SECRET) to the less sensitive information (e.g.

CONFIDENTIAL). Such a case could occur and is termed the *cascading problem*, which is described in Appendix C of the Trusted Network Interpretation of the TCSEC (TNI) [24].

4.4.3 What to Look For

Evaluators should examine the integration process to ascertain that this process identifies all the necessary aspects of building the trusted system from its components. Evaluators should be confident that the integration process can reproduce a previous build if it is required. This includes any environment variables and COTS software. The evaluators should also be confident that the integration process controls the build so that extra components are not included.

4.5 Integration Testing Process

The integration testing process is the process of attempting to find defects in the integrated capabilities built by the integration process. The integrated capabilities should be chosen with an eye towards building the test suite for the security features. This may mean including some of the unit test cases of the security relevant components (e.g., the components implementing the security mechanism) into the integration test suites. Integration testing is "black box" testing which only knows about the public aspects or interfaces (e.g., inputs and outputs) of the units. This type of testing therefore tends to find design defects. The evaluator needs to assess the integration testing process to ascertain that this process had detected and corrected any design defects that may have initially been in the product(s). (Note that an evolutionary development process may utilize the strategy of integration testing components to iterate to a final design of the system.) Integration testing may also find deficiencies in the unit testing process that allow implementation defects in the integrated software to remain undetected. Therefore, the evaluator needs to assess the defect data collected by the integration testing process to ascertain whether the unit testing and integration testing combination was effective at removing implementation defects to the point that few defects are likely to remain.

Integration testing should occur in its own environment (e.g., could be another root directory, node, or platform, etc.) separate from the development environment. In general, integration testing should occur on the target platform and not a substitute. Code and data, including test data, should only arrive in the integration environment through the configuration managed integration process or through the execution of the code. Only the integration testers should be able to access the integration testing environment. Evaluators should be confident that the integration testing process is a controlled, repeatable process that tests capabilities built from known versions of the integrated units in a known environment. If the integration tests are not performed in this manner, it may be difficult for the evaluators to conclude that the integration testing and rework cycle have removed all of the design defects from the trusted software.

Integration testing should include module coverage testing and stress testing. Depending upon the trusted system being developed, there may also be a need for the integration test procedures to include one or more of the following:

- concurrency testing and
- COTS usage tests.

Each of these types of integration testing will be explained in more detail in following subsections. Further details can be found in [2], [3], and [22].

As a result of integration testing, defects are found and corrected. The integration tests need to be rerun on the corrected code. This regression testing is benefited by the automation of the execution of integration test procedures and the comparison of outputs to

required outputs for the test. With automation of the integration test procedures there is less tendency to minimize the amount of regression testing on corrected components, and hence there is more assurance that the corrections did not cause any unpredicted side-effects or defects. Regardless of the level of automation of regression testing, when the source for an integration capability (e.g., a CSC) is modified, the associated integration tests need to be rerun to verify the correction and to verify the lack of injected defects.

4.5.1 Module Coverage Testing

Module coverage testing can be performed at three levels:

- C0 : All modules are executed at least once.
- C1: All pairs of modules in the modular invocation graph (i.e., a graph of which modules other modules call) are executed at least once.
- C2: Every call is exercised at least once.

Typically, C2 coverage where all branches are exercised is too time consuming, so most module coverage is planned for the C1 level of coverage.

4.5.2 Stress Testing

Stress tests evaluate the CSCI or system under a heavy load to saturate one or more of the resources. Thus, stress tests target errors in resource contention and depletion and errors in limits, thresholds, or controls designed to deal with overload conditions.

4.5.3 Concurrency Testing

Many systems need concurrency testing as part of the integration testing. The objective of this kind of testing is to look for the following errors in performance and timing:

- Resource contention - multiple processes requiring the same resource;
- Deadlock conditions - where processes "A" and "B" both need resources "1" and "2"; process "A" grabs resource "1" and is suspended; process "B" grabs "2" and cannot get resource "1";
- Race conditions - process "B" uses variable "A" before "A" has completed its initialization or process; or "A" is scheduled and defines "temp" but before "A" references "temp", "B" is scheduled and redefines "temp"; and
- Synchronization - a process is waiting to synchronize with another process which is unscheduled, or terminated, or process "A" writes over data base variable which is assigned to process "B."

Concurrency tests should be based upon scenarios of the expected operation of the trusted system, otherwise, there may be no end to this type of testing. Also note that these tests are only valid when performed in the target environment.

4.5.4 COTS Integration Testing

Thorough testing of COTS products is not always feasible, but such products need to be tested under their conditions of use to a sufficient extent to determine that they are not contributing defects to the trusted system. Constraints for how a COTS product is used must be documented and understood. If a COTS product provides functionality that is not desirable, then the constraints on the COTS product usage must be determined, documented (in maintenance manuals or the user manual for system administrators), and enforced.

4.5.5 What to Look For

Evaluators should be confident that the integration testing process is a controlled, repeatable process that tests capabilities built from known versions of the integrated units. In general, this requires a separate integration environment on the target hardware system with access limited to the integration testors.

Evaluators should be able to examine the integration test procedures and results to satisfy themselves that module coverage tests and stress tests were performed and that no uncorrected anomalies or defects resulted. The stress tests and concurrency tests should have included the maximum or exceeded the maximum load for the expected trusted system to make sure that the system behaves properly under heavy load. The integration testing process should have also included COTS testing as appropriate and defined within accepted test plans.

Evaluators should be able to examine the integration test records and determine that the integration tests have detected defects and that these tests have executed successfully following the defect correction. They should then be able to conclude that the integration testing and rework cycle has removed all the design flaws that may initially have been present.

4.6 Documentation Process

The documentation process as used in this chapter produces user documentation. Software documentation (e.g., SDFs, Version Description Documents) is excluded from this process as it is part of the other implementation and integration processes. Two issues associated with the documentation process are the need for standards as well as the need for inspections that include coders.

The documentation process requires clear standards and inspections to ensure the adherence to those standards. The documentation process usually produces detailed documentation standards, guidelines, and templates to aid consistency and readability of the documents. These are in addition to company standards or data item description (DID) formats. For example, there should be a glossary of key words (and their spellings) specific to the implementation for the documenters.

The inspection of documents such as user manuals should include the implementer of the software to ensure the accuracy of statements made about the software. However, user documents must be written from the point of view of the user — not from the software structure point of view — so users can understand them. This often requires the documentation writers to be in contact with the final customer/user of the software. Rework and action items identified by the inspection of the documents must be tracked to completion and verification.

Documentation writers should have access to reasonable stable, integrated versions to verify their assumptions about how the software looks to a user. Often a clone of the integration environment and integration test data provides enough of the basics for documentation writers. The writers are also likely to require capabilities for screen capture, etc., that may not be part of the final product or system.

The documentation process should have provided documentation standards that are understandable by the evaluators and that are, in addition, required formats or DIDs. The evaluators should have confidence that the documents accurately reflect the trusted software and its end use in an environment. The documents should have been inspected for this accuracy and for adherence to the documentation standards as a normal part of the documentation process. This process should also have verifed the user documentation against the assumptions and assertions for the user environment of the trusted system.

4.7 Support Tool Development and Use

Support tools, particularly automated support tools, are beneficial to all of the implementation and integration processes. Consideration should be given to providing support tools to automate any of the repetitive tasks but particularly the error-prone tasks that have well-defined rules. Many support tools are readily available commercially. Proper trade studies and comparisons can select the most useful of these tools to support each process. There may then remain the need to integrate several tools to provide the automation desired.

Evaluators should undertand how these tools were used in support of the implementation process. They should become familiar enough with the tools to understand the limits of the tools. It should also be noted that a lack of automated tools does not exempt a project from performing the necessary functions of the implementation process. Use of a specific tool may require the adjustment of some the company's standard process or standards. This use, however, should not be an excuse for practices or standards counter to guidelines in this chapter.

As stated in section 2.1, there is a direct TCSEC requirement for a configuration management process and supporting tools. This is discussed in the next section.

4.8 Configuration Management Process

The configuration management process must accept the inputs from the implementers and use these sources to build the integrated code for the integration process and to release code successfully through integration testing for system testing. The CM process must also accept inputs from testers and documenters. Ultimately, the CM process builds releases of code and documentation for trusted distribution.

The CM process provides the following main functions:

- configuration identification,
- configuration control,
- status accounting on configuration(s),
- configuration audit, and
- release or distribution of configuration items.

Configuration identification is necessary for configuration control. Control of changes is the core function of CM. In fact, CM exists because changes to an existing or a developing system are inevitable. The purpose of CM is to ensure that these changes take place in an identifiable and controlled environment and that they do not adversely affect the system or the implementation of the security policy of the TCB. Control necessitates the status accounting of what has been authorized to be changed, what the progress of that change is, and then auditing to ensure that the change was made properly and nothing else was changed by accident.

The CM process outputs change proposals, minutes or reports from configuration change boards, and reports of the current status of configuration items in addition to the releases.

Evaluators need to be familiar with concepts and issues associated with each of the main functions of CM. These concepts and issues are discussed in the following subsections [25].

4.8.1 Configuration Identification

The CM procedure should enable one to identify the configuration of a system at discrete points in time for the purpose of maintaining integrity of the system and traceability of the configuration throughout the system life cycle. The configuration identification scheme should be described at the beginning of the project in the CM plan. (Note that the CM plan can be part of a system engineering plan or program plan and can reference company CM standard practices and procedures.) New items may be identified during implementation and integration. Therefore, the configuration identification scheme has to be extendible. The configuration identification of the items such as CSCIs, CSCs, and CSUs in the system, should be mostly an accomplished fact by the implementation phase. This part of the unique identifier usually represents the place of the item in the system by values in certain of the subfields of the identifier. In addition, the version of the item needs to be a subfield so that a change to the item produces a new configuration identifier.

CM identifiers must also be applied to the documentation and test items. It is helpful if these identifiers can be easily related to the software and hardware configuration items they depend upon and vice versa.

Configuration identifiers need to be compatible with the source control facility utilized by the CM process. Several automated tools are available for controlling sources and providing version identifiers (e.g., Unix SCCS, VAX DEC/CMS, etc.). These tools provide the required ability to retrieve previous versions in addition to the current version and to show the difference between versions.

4.8.2 Configuration Control

The TCSEC mandates a CM system for B3 systems to maintain control of changes to the following items produced during implementation and integration:

- source code,
- "the running version of the object, code" (e.g., executables),
- implementation documentation (e.g., user manuals, operating procedure manuals),
- test "fixtures" (e.g., test data, test procedures and scripts, test drivers, unit test cases, etc.), and
- test results.

The TCSEC also requires the CM system to provide tools for comparisons of newly generated TCB version with the previous version to ascertain that only the authorized changes have been made in the new version.

Configuration control needs to approve all changes and monitor their status from inception through implementation and testing to release. Authorizing needed changes is usually performed by a CCB (configuration control board or configuration change board). This board is composed of qualified members who typically represent the different functional areas of the program, including security engineering. Typically, the head of the CCB is charged with communicating the change proposals to be discussed at the next CCB meeting and ensuring that there is adequate representation to analyze the proposed change. The purpose of the CCB is to evaluate proposed changes for impacts. It is important to have a cross-section to prevent contradictory changes and to provide the best chance of detecting potentially negative side-effects. The CCBs should produce records or minutes of the analysis and a definitive status for the proposed change (e.g., authorized, rejected, or postponed for further study). The CCB also ensures that all configuration items affected by the change have been identified.

4.8.3 Status Accounting

The purpose of status accounting is to record and report anything of significance to the CM process. What will be recorded, how it will be recorded (e.g., in a data base), and how information will be reported as respecified in the CM plan. It is usually beneficial to have an on-line status accounting system as it is less cumbersome to query a data base of structured information than to look through documentation to ascertain the status of a configuration item. Whatever system is used, it should be possible to locate the authorized versions of any configuration item and determine the current status of that configuration item.

Status accounting should produce periodic reports on the current configuration(s), the status of configuration items, and the status of authorized changes. Status accounting should also be able to provide a complete historical list of proposed changes and their status.

4.8.4 Configuration Audit

Configuration auditing involves inspecting the configuration status accounting information and verifying it against the actual configuration items. This auditing is to verify that the CM process is working correctly. For trusted systems, it is important that configuration audits be performed periodically to minimize the chance that unauthorized changes have been made and not noticed. Accurate status accounting information provides a demonstration that CM assurance is valid.

4.8.5 Release of Configuration Items

Part of the CM process is the process of releasing configuration items such as the trusted software executables and the corresponding documentation. This release process should only release items authorized for release. There may be several types of release such as release to the integration environment for integration testing, release to an environment for system testing or security testing, and release of documentation for reviews or audits, etc. These different types of release share common procedures but may have different authorizing personnel and criteria.

The CM process needs to build these releases for the appropriate target(s) and on the appropriate media from the master library of configuration controlled items. This master library is central to the CM process. It is typically in an automated facility such as Unix SCCS or VAX DEC/CMS. Only limited discretionary access should be provided to the master CM library to ensure that it can not be corrupted. This assurance is necessary to ensure that the release process can be trusted not to contain any "extra" or malicious code.

Concurrent or parallel changes to the same configuration item can complicate the CM release process. Multiple target environments, each requiring its own "post linking," can also complicate the release process. If either of these complications is present, the procedures for handling that aspect of the release need to be particularly examined to ensure that the CM release process does what it is supposed to do without errors.

4.8.6 What to Look For

Evaluators should understand the configuration identification scheme used by the configuration management process. This scheme should be in accordance with the CM plan. Evaluators should also understand the tools used in the configuration management process to control the configuration and provide historical versions of the components of the trusted system. These tools must provide the ability to compare versions to ensure that only authorized units and authorized changes have been incorporated into the source library. The configuration management process must also insure that only items maintained in the sources library are used in the build and release process.

Evaluators should look at the change control records. These records should show clearly what was changed, who was responsible for each change, that the changes were authorized, and that the authorization process studied the rationale for the change and the impact of the change. The impact on security and the assumptions and assertions of the affected components should have been studied. The change notification process should show notification of all affected parties and it should check for closure on the change (e.g., all affected documents and tests have been updated for a software change). The status of each change should be known.

The change authorization process should have occurred according to the SDP. Evaluation of changes is best done prior to making the changes, but it is acceptable if some of the changes are documented and studied after the fact. However, evaluators should check that the changes, rationales, and impact studies are understandable, reasonable, and necessary and not a "rubber-stamp" of as-built code. The change authorization typically involves a a technical assessment by a configuration board of several people representing specialty areas (e.g., security, testing, data base design, documentation, etc.) in addition to the technical lead. It is important that enough people have reviewed the change for negative impacts so that the likelihood of having missed any is low, and also that the change has been reviewed for impacts upon the security assumptions and assertions.

5 ASSESSMENT METHOD

This section presents a suggested method for an evaluation team to use when addressing an actual implementation. As identified in the introduction, this chapter assumes that the design was evaluated and found acceptable at a B3 level (perhaps following rework). It also assumes that the plans (management plan, SDP, test plan, etc.) were accepted and maintained.

The assessment method gathers information on the implementation process and the implementation products. The method utilizes the analysis of the implementation subprocesses to identify the items most likely to have defects. Thus, this method benefits the evaluation team by providing a focus for the evaluation of a potentially overwhelming amount of material.

5.1 Preparation

The following information should be gathered:

- project plans such as management plan, software development plan (SDP), configuration management plan, software quality evaluation plan. Note some of these plans may be combined,
- design documentation, including the DTLS, security architecture information, policy model,
- source code,
- Software Development Folders or Software Development Files, Unit Development Folders (UDFs), Software Development Notebook, or programmers notebooks,
- code review minutes and follow-up (in the SDF),
- Configuration Management records (e.g., Configuration Control Board minutes, change proposals, version or build descriptions or lists, reports),
- test plans, test procedures, test cases, and test results,
- defect or discrepancy reports {typical names are Discrepancy Report (DR), Software Discrepancy Report (SDR), Software Problem Report (SPR), Software Trouble Report (STR)}, and
- management indicators such as module completion dates, inspection/review dates, staffing curve, etc.

An overall survey of this information should indicate that the information is ready, available, and visible to the developers and evaluators. Much of this information, while critical to an effective evaluation, will typically not be in the form of deliverables under an associated contract. To obtain the information, the evaluation team will need to rely on the cooperation of the developers. On the other hand, the developers rely on the goodwill of the evaluators, which they are likely to lose if they attempt to trick the evaluators or insult their intelligence.

5.2 Observation of Readiness to Evaluate

Evaluators should look at the implementation processes before analyzing the implementation products. There is no point in analyzing the implementation products if the implementation is not complete. Similiarly, there is no point in studing the source code, if the evaluators do not have confidence that the project control and configuration management processes have controlled the configuration. The assurance that the source

code provided to the evaluators is in fact the source for the trusted system that would be released is essential for continuing the evaluation process.

Management indicators, metrics, trend analyses, and/or QA reports should be available as specified in the management plan, SDP, and QA plan. Evaluators should verify that the implementation followed the plan within an acceptable level of deviation. The experience level of the developers and inspectors should be reasonable and according to the project plan. Module completion and inspections should have occurred roughly as planned.

Configuration management records must be adequate to insure that the source code obtained for evaluation or inspection is the source code that was tested and that all of the necessary sources for building the system are under configuration management control. "All necessary sources" includes data for data-driven software, compilation/link order or *make* files, and environment variables or logicals. These records should show that changes to the design and implementation were performed in a controlled, well thought out manner and were tracked to completion. Changes to the baselined, accepted design (or baselined portion of the design in an evolutionary development life cycle) should be minimal. Configuration Management records should indicate that any changes were studied to ensure that the change would not have any undesired side-effects and would not impact the security features. These records should show that proposed changes were communicated for review, that only approved changes were implemented, and that approved changes were tracked to completion. There should not be any evidence of back-filling the change proposals to fit an as-built system without due consideration of the impacts of the changes.

There should be SDFs (or the equivalent) for each of the units or modules of the system being evaluated. Ideally, these should show evidence of being used by the developers and not that they were generated as an after-the-fact documentation chore. SDFs should contain records of the reviews, inspections, etc., specified in the SDP. Records indicating the resolution of defects identified in these inspections may also be included in the SDF or may be separate.

There should be adequate test reports, regression test corrections, and discrepancy report (DR) documentation. There should also be records from the tracking system used to ensure the correction of the identified bugs. Evaluators should have confidence that all identified defects have been tracked and have been corrected as a result of this documentation.

If appropriate, the minutes for the Test Readiness Review or equivalent should show that modules had completed unit-testing and appropriate reviews before integration and integration testing. Completion of most of the modules at the very end is a negative indicator and suggests that some of these last modules may have been rushed and therefore contain defects. Evaluators should spend extra time analyzing these "last-minute" modules.

Note that CM records, SDFs, DRs, test reports, etc., may be kept on-line in the development system or may be in hard copy form. The form and directions for filling out the forms should be given or referenced in the appropriate plan. The evaluator should be able to obtain these directions readily as the directions are usually necessary to interpret the data on the forms.

5.3 Analysis

The purpose of an analysis of the software implementation is to ensure that the intent of requiring a modular system (a system with referential transparency) is not subverted by a poor implementation and to determine that there are no remaining design flaws, no more than a few correctable implementation flaws, and to establish a reasonable confidence that few flaws remain.

The analysis of the software implementation involves analysis of the software development process used in implementation and analysis of the implementation products. The analysis of the process ensures that the development process was a controlled process by verifying that the procedures identified in the software development plan were followed, the change control process was followed, and that the planned reviews and testing occurred. These procedures and reviews should have prevented defects and any extraneous insertions of code or functions. Analysis of the products ensures that the implementation is a good implementation that has been adequately inspected and tested such that few defects are likely to remain.

5.3.1 Analysis of Software Implementation Process

Analysis of the code review documents ensures that implementation standards were addressed in the reviews. These standards should be uniformly applied to the code. The types of implementation standards added to the general coding standards reflect the specifics of the developed system and its environment. Typically, implementation standards identify common ways to detect and handle errors, to test status from hardware or other system calls, and to test status from data base calls, etc. Standards for commenting and language feature use and prohibition should also have been checked. Automated checks (e.g., via tools like *lint* for C, some of the compiler options available in Ada compilers) are more desirable than manual checks, but either type of checking is acceptable.

Cluster Analysis or Pareto Analysis [16] of the defects found in the implementation process identifies the modules in the product that need extra attention and may need re-implementation. Analysis of the defect causes (e.g., lack of initialization of variable, boundary logic, use of wrong data item) identifies probable defects for which to reinspect the code and may identify weaknesses in the implementation and review process.

Evaluators should perform the Cluster Analysis on the units and the defect causes if these have not been provided. If these analyses have been provided, evaluators should verify them. Evaluators should use the results to identify if there are any units or any error types that need extra attention during their analysis of the implementation source code.

5.3.2 Analysis of Software Implementation Products

Analysis of the implementation products ensures that the modularity of the design and the modularity intent (i.e., readable, understandable, testable code) has not been subverted by poor coding/implementation practices such as :

- creating modules that are too large, too complex, or insufficiently commented,
- improper scoping (localization) of variables,
- improper use of low level features including hardware modes,
- insufficient error detection and/or error handling,
- addition of entry and exit points to modules, and
- addition of extra functionality breaking the cohesion of the module.

The rationale for looking at these implementation practices is given in the following subsections.

5.3.2.1 Size and Complexity

Size and complexity both influence the understandability of modules and hence the understandability of the system. Complexity has two forms: (1) difficult to understand and expensive to implement (e.g., spaghetti code) and (2) easy to implement but difficult to

understand (e.g., recursion). The first type of complexity must be avoided in the implementation; the second type can be improved with the appropriate annotation of the code. While the evaluators' judgment must be used to decide if the implementation is understandable, size and complexity measures will usually substantiate this judgment. Why measure the size and complexity of code? Size and complexity of a module are both highly correlated to the number of defects in the module [27]. Size and complexity of all modules in the system should be measured and should, in general, be within the guidelines specified in the coding standards (part of or referenced by the software development plan). These standards will probably be domain specific to the language utilized and to the type of application. Complexity may be measured by hand or by automated methods. Typical measures of complexity are :

- Information content as measured by the Halstead Software science [14] that regards computer programs as coded messages to the computer, the information content of which can be measured and used to predict the effort needed to develop the program and to estimate the defects that will occur in the program. Modules that exceed the guidelines in the development standards need to be evaluated to determine if they are too complex.
- Function Points [13] that measure the functionality delivered to the user. Though the exact function point numbers may vary between organizations, function point analysis has a strong correlation to the lines of code, effort, and the Halstead numbers. If the module size exceeds the predicted size by very much, this may mean that the module is more complex than it needs to be or provides extra functionality.
- Information Flow metrics [15] that measure the fan-in and fan-out of procedures and then correlate the function (fan-in * fan-out) **2 with the probability of the procedure needing a correction or change. The correlation for this metric with the need for changes to the procedure measured was much higher than the correlation with the length of the procedures.
- McCabe Cyclometric complexity that was originally designed to measure the number of "linearly independent paths" through a program and is highly correlated to testing effort, maintenance effort, and defect counts. Most organizations place an upper limit of 10 to 20 on the McCabe cyclometric complexity index. Modules with a higher index need to be evaluated to see if they are unnecessarily complex and to see if they have been adequately tested [20].

In addition to the common measures of complexity, object oriented measures are being developed [18] and [19] because some of the common measures of complexity are not adequate for object oriented techniques. Thus complexity might be measured in terms of metrics similar to the following:

- Depth of Inheritance Tree - a measure of how many ancestor classes can potentially affect a class. It is useful to have inheritance for reusability, but if the inheritance gets too deep, it may be hard to test.
- Number of Children - a measure of how many subclasses are going to inherit the methods of a parent class. This measure givers an idea of the potential influence a class has on the design or implementation. If a class has a large number of children, it may require more testing of the methods of that class.
- Weighted Methods per Class - tries to relate the complexity of the behavior of a class by summing the cyclomatic complexities for each method in the class. This should account for the fact that a class with many simple methods may be a difficult to understand as a class with only a few complicated ones.

- Coupling between Objects - a count of the number of non inheritance-related couples with other classes. Excessive coupling is detrimental to modular design. This measure is useful to estimate how complex the testing of various parts of the design will be. A high degree of coupling between objects has been correlated with high defect rates.
- Response for a Class - the response set is a set of methods available to the object and hence is also a measure of communication between objects. If a large number of methods can be invoked, the testing and debugging of the object becomes more complicated and more subject to defects.
- Lack of Cohesion in Methods - uses the degree of similarity of methods. Cohesiveness of methods within a class is desirable to promote the encapsulation of objects.

Techniques such as the use of generics, renaming, and overlay of operators in Ada can improve understandability and decrease complexity and code size or can be abused and actually decrease understandability The tendency of C coders to condense the code as much as possible also decreases the understandability of the code.

Evaluators should give the most complex units the most attention. The purpose of this attention is to evaluate the understandability and correctness of the code and to ensure that nothing unexpected or malicious has been slipped in. If evaluators cannot understand the unit with reasonable assistance from the other documentation and the development team, then the unit must be re-implemented.

5.3.2.2 Scoping or Localization of Variables

The variables in the implementation should be localized to the lowest level procedure or function requiring that variable. Variables should not be global but hidden from other procedures to the maximum extent possible. Penetration attacks often rely on the misuse of variables or pointers; therefore, good information hiding or encapsulation in the implementation limits the opportunities for penetration attacks on the system. Evaluators should question the scope of any variables which seem to be global or have a scope wider than apparently necessary.

Language features such as "private" types, "input" or "output" parameters instead of "input_output" parameters in Ada should be used to provide the appropriate scope for variables. "Include" in C or "imports" should only make visible the minimum variables required by the module. Local copies of validated inputs should be used in reference to rereading an input source that may have changed. Passing of pointers should be avoided as penetration techniques often use reassigned pointer values to accomplish the penetration. Evaluators should understand all exceptions to these guidelines and be confident that these exceptions do not provide an exploitable vulnerability.

5.3.2.3 Error Detection, Error Handling, and Interrupt Handling

The implementation must address error conditions or unexpected conditions. Assumptions of reasonable behavior in calling the modules should not be made. Features such as exceptions in Ada or hardware under or overflow conditions, essentially generate interrupts for unanticipated conditions. All modules must be robust enough to deal with these potential conditions. Penetration attacks often rely on providing unexpected inputs or interrupts which then puts the application or system in an undetermined state or a state which denies service to other processes. Repeated use of these conditions can be a way to systematically determine enough about the system to penetrate it.

Similarly, the hardware instruction set or hardware mode used in the implementation should be the minimal set needed to provide the function. Thus utilities for a user must not have direct access to the protected or kernel mode.

5.3.2.4 Additional Entry and Exit Points

Addition of entry and exit points to modules during implementation can also break the security assurance of the design. Any additions should be re-evaluated to insure that the implementation is still consistent with the underlying security policy and security design.

5.3.2.5 Extra Functionality and Extraneous Code

In non-security-related development projects, often it is desirable to provide extra functionality in the implementation particularly if the extra functionality does not take more effort. Unfortunately, adding extra functionality to trusted software is not desirable for the following reasons. First, extra functionality may increase the size or complexity of the code. Second, the extra functionality may decrease the functional cohesion or the understandability of the module. More significantly, the extra functionality may hide a back door or malicious code or provide extra information that can then be used to penetrate the system. Thus evaluators need to analyze the implementation to ensure that nothing has been added to the design unnecessarily.

Extraneous code is code that serves no useful purpose in the evaluated product. Examples are functions that are never invoked, code that is circumvented by the logic, functions that simply return when called (and the associated call), unused data structures, variables, and type definitions. Extraneous code generates a distraction and thus hampers the understanding of the system. Extraneous code also poses a hazard for software maintenance because changes could be made to invoke this extraneous code which has not been tested. Clearly, extraneous code should be detected and eliminated. Evaluators should see that this detection and removal have occurred as part of the development and review process.

In conclusion, this guideline notes that providing extra functionality appears to be an area where the concept of good software engineering may be counter to basic security principles and design. In some application domains adding functionality "for free" may be considered desirable. An evaluation team will have to aware of the potential "cultural difference" that may exist here.

5.3.3 Analysis of Test Procedures and Test Cases

Analysis of the implementation products also needs to ensure that adequate unit testing and integration testing have occurred by:

- inspection of the test cases and test procedures,
- inspection/analysis of test records to determine if all branches (possibly supported by coverage analyzer) and most paths (combinations of branches) have been covered by testing,
- comparison of complexity, functionality estimates versus the number of test cases for the module, and
- comparison of estimates of probable defects (e.g., Rayleigh model, [27] past development of similar product) versus the number already detected and corrected.

As mentioned earlier in this chapter, the testing philosophy differs between testing a trusted system versus a low assurance system. For a trusted system, testing is the process of executing a program, or program unit with the intent of finding errors. Testing is NOT just demonstrating that the program works. Units or modules are unit tested by using test cases. After the units have passed the unit test cases, they can be integrated and further tested using test procedures. Typically, unit testing finds defects in coding the unit, while integration testing can find defects in the coding and design (particularly in the interfaces between units).

5.3.3.1 Unit Testing

Unit testing tests each unit to determine whether the unit has met its functional specifications (i.e., requirements which may include derived requirements for performance, timing, memory usage, etc. in addition to the service it provides). Test cases need to specify the expected output or result for the given inputs. A good test case is one that has a high probability of detecting an error. Test cases need to include invalid and unexpected input conditions as well as expected valid inputs. Test cases need to include the boundary conditions because the boundary conditions (or limits of the parameter range) are where most of the coding errors occur. The results of each test case need to be thoroughly evaluated. It is important to determine not only that the code produces the correct results within the required criteria but also that the code does not do something it is not supposed to do. It is preferable to have a knowledgeable person other that the developer of the code develop the test cases and evaluate the results.

To the extent feasible, test cases should be scripted so that regression testing can be automated. Thus when an error is corrected, the test cases for that unit should be rerun to ensure that nothing was broken by the fix as well as verifying that the fix corrected the whole error. Without automation of the regression testing, many defects introduced by the correction of other defects can slip through, since ad hoc testing by the person making the correction is rarely systematic and complete enough.

In addition to the typical unit test case testing, sometimes referred to as equivalence partitioning and boundary value testing, it is also helpful to include data flow analysis and data base analysis in the code inspection.

Data Flow Analysis is a process of examining the definition or assignment of variables and the uses of variables in the code and searching for anomalies such as:

- variables assigned but not used,
- variables used but not assigned,
- variables assigned twice without use in between, and
- unreferenced variables (declared but not assigned or used).

Some apparent anomalies may be necessary (for instance, double definitions if a routine is sending data over a bus), but most anomalies should be investigated.

Data Base Analysis is inspecting the code (or design) focusing on the compatibility of data base variables throughout the code. The variable usage should be consistent with the data dictionary. This inspection primarily checks the variable types, scaling of variables, and precision of variables as used in the code.

5.3.3.2 Integration Testing

After the units have been tested and have successfully passed the unit test cases, they can be integrated into configuration items such as CSCs, CSCIs, TCB, etc., and integration testing can start. Integration testing is performed to determine whether components that are individually satisfactory are consistent and correct when combined. A few examples of integration errors are inconsistent data validation criteria, inconsistent handling of data objects, and timing errors. Integration test procedures should include module coverage testing and, in general, must be performed on the target platform. Typically, module coverage where all branches are exercised and tested is too time consuming, so integration testing makes sure that all pairs of modules in the modular invocation graph are executed at least once. The integration test should be performed to the level specified in the accepted test plan.

Integration testing of many systems needs to include concurrency testing. The objective of this kind of testing is to look for resource contentions, deadlock conditions, race conditions, and synchronization problems that might occur in the multi-user operational environment. These tests are valid only when performed on the target system environment.

Stress testing should also be a part of the integration tests. Stress tests evaluate the CSCI or system under a heavy load to saturate one or more of the resources. Thus stress tests target errors in resource contention and depletion and errors in limits, thresholds, or controls designed to deal with overload situations.

In addition to seeing that the appropriate types of integration tests were performed, evaluators should see a decline in the number of defects in the integration test records over time. (This is also an increase in the mean time to failure of the integration tests over time). Evaluators should not see evidence of "thrashing" (i.e., where one error is corrected but causes another error and then fixing that error negated the first fix, etc.). "Thrashing" problems are usually the result of design defects or lack of configuration control, neither of which is acceptable for B3 trusted software.

5.4 Interaction

Evaluators may need to interview and meet with developers to clarify

- coding standards,
- explanations or alternatives for exceptions made to guidelines,
- code that is intrinsically complex (e.g., unique algorithms, recursion), and
- discrepancy resolutions.

Interviews and meetings should be documented. Clarification requests and responses should be maintained. These materials can aid any future recertification efforts and provide a historical perspective of the evaluation effort.

5.5 Reporting

The evaluation assessment report should be organized for clarity, effectiveness, and support of the conclusions drawn. A possible outline is to go through each security requirement and provide:

- requirement statement;
- applicable features of the implementation relative to the requirements (e.g., this could be provided by a mapping of software modules to the security requirements.);
- assessment of those features including:
  - areas for recognition,
  - areas for improvement or areas of concern,
  - areas for additional review,
  - summary conclusion indicating whether the requirement was met by the implementation,
  - understandability, and
  - complexity;
- assessment of the testing and documentation of those features; and

- overall recommendation.

Evaluators should distinguish in their assessment between the quality of the software and user documents versus the quality of the documentation about the implementation. It may be easier to correct poor documentation of an essentially sound implementation than to correct a poor implementation.

6 ASSESSMENT EXAMPLE

This section provides an example of the assessment method as described in Section 5. This is an actual worked example, an assessment of an internet system performed by NCSC, NSA, and MITRE. That assessment was completed in June 1989, in support of a possible TCSEC A1 certification. The focus in this section is on the B3 aspects of the assessment, specifically in the area of implementation evaluation. Therefore, the complete MGS assessment report [26] is not reproduced here.

The system addressed by the example assessment is the Multinet Gateway System (MGS), developed by Ford Aerospace and Communications Corporation. MGS is an internet system of gateways, supporting interoperability of dissimilar networks, internet protocol (IP) routing, and information security (INFOSEC). INFOSEC support is based on the IP security option field (mandatory for MGS use), on a specialized Gateway Authentication Protocol (GAP), and on encryption of traffic over public transport networks.

This section is presented in an organization that matches that of Section 5.

6.1 Preparation

Section 5.1 indicates a variety of information needed for a system assessment. That information was generally available to the MGS assessment team as described below.

Much of the information provided was in the form of technical deliverable items, including:

- project plans: Configuration Management Plan, Software Quality Assurance Plan, Software Development Plan, and Software End Product Acceptance Plan;
- design documentation: Software Requirement Specification, Interface Specification, Software System and Subsystem Specification, Software Program Specification, and System Security Description;
- source code; and
- deliverable management reports and meeting minutes.

Further, because of the open working relationship among the customer, the contractor, and other reviewers, considerable additional material was made available on an informal basis. This material included unit development folders, code review minutes, configuration management records (both on-line and off-line), discrepancy reports and dispensations, and various management indicators.

It is worth noting that if the assessment team had to rely on the deliverable items alone, the assessment would not have been very effective. In fact, the developers maintained a very open configuration data base, with the evaluators free to examine their on-line as well as paper organization and information. For example, even though the developers had originally expected that the evaluation team had the need to examine only TCB code in detail, the evaluators were uncomfortable with this limitation. The evaluators felt that they should also examine untrusted application code to assure themselves that none of it needed to be part of the TCB. As a consequence, the developers opened up all their source files (on-line and hard copy) to the evaluators. It is of course ideal to have a good working relationship so that such an open environment exists, but it is also a good idea to indicate up-front that such information needs to be available for an effective assessment.

The assessment team did obtain and survey all this information as it became available and used the information as the basis for its assessment.

6.2 Observation

An important element of the observation aspect, as indicated in Section 5.2, is the comparison of the project activity with the specified plans. This was accomplished by the evaluators studying management and technical plans, reviewing activity records (such as review minutes and configuration management records), attending some meetings at the developer site, and conducting other program review meetings. The information thus obtained was compared for consistency and reviewed to determine the adequacy of the project in terms of an A1 (and therefore also B3) development.

Part of the observed information was on-line change tracking produced automatically (by Unix's SCCS) and maintained in a way that "backfilling" (e.g., by alteration of modification time stamps) would have been virtually impossible. By observing a combination of information both formally and informally available, the validity of the body of information was established.

The results of the observation aspect of the assessment were documented in Sections I through VII of the MGS Final Design Assessment Report. Those sections (making up about three-fourths of the report) are a summary of the available information (of all kinds). That summary provided a sufficient basis for the analysis aspect.

6.3 Analysis

The analysis aspect of the assessment of MGS is the final one-fourth of the Final Design Assessment Report [26]. This portion is composed of Section VIII (TNI A1 M-Component Evaluation), Section IX (Conclusion), and Appendix A (Evaluator Comments).

6.3.1 Analysis of Software Implementation Process

Because the implementation was evaluated only against the TCSEC and TNI requirements, configuration management was the only requirement area applicable to the implementation process. The configuration management plan and tracking information were examined and found to be satisfactory. The assessment included a description of configuration management as planned, comparing the plan to the actual configuration management activity on the project.

The analysis of the configuration management process concluded with the following observation:

> *Two aspects of Ford's configuration management are especially noteworthy. Being a government project, configuration is more extensive and rigorous than what is typically found in commercial developments [26 p. 162].*

Thus, the analysis included three aspects: examination of plan, examination of activity, and conclusion.

6.3.2 Analysis of Software Implementation Products

The implementation work products were analyzed along with documentation that relates them to particular criteria requirements (particularly the code correspondence analysis). Since the implementation was evaluated only against the TCSEC and TNI requirements, correlation of the implementation with the design was the only requirement directly applicable to the implementation product.

The developers had performed this correlation in a somewhat non-standard manner. Ordinarily, a code correspondence analysis (CCA) is performed relating the TCB code to

the descriptive top-level specification (DTLS). The developers had broadened this effort considerably, in that they performed an implementation correspondence analysis (ICA), which compares the entire implementation (all software and hardware) with the specification. The evaluators reviewed the ICA as given, documenting it at a detailed level, and considering the validity of the results in terms of giving the desired correlation. Finally, they offered the following conclusion:

> *These informal arguments go beyond what is usually done for code correspondence analysis, and they involve an intriguing combination of human and machine effort [26, p. 110].*

Since neither the TCSEC nor TNI considers quality of the implementation work product itself (as described in Subsection 3.2), the report lacks any commentary on that subject. That was necessarily appropriate, since the applied criteria do not deal with the subject, and since evaluation resources were limited and needed to be applied to subjects that were dealt with by the criteria. As a result, certain shortcomings in the code that the developers observed in retrospect remained.

For example, the corporate software standards insisted that the first three characters of every name within a software unit be a short code for the name of the unit, in order to avoid collision in the global name space. This was both unnecessary and in fact detrimental. It was unnecessary because the linker catches any such collision. It was detrimental because that naming convention significantly reduces readability of the code. If the evaluators had a mandate to deal with such issues, their observations might have constituted a force to obtain an exception from the standards in that case, and possibly even to revise the corporate standard.

6.3.3 Analysis of Test Procedures and Test Cases

Security testing was found by the evaluation team to be beyond the scope of this particular evaluation effort:

> *The assessment team performed an analysis of the source code, but did not perform security testing. Although the evaluation of the security testing was not in the scope of the assessment team, Ford Aerospace did accomplish a testing program that addressed security requirements. [26 p. 145]*

Nevertheless, the testing documentation was examined and found to meet applicable criteria:

> *The test plans . . . procedures and results [TEST] provided satisfy the intent of the TNI criteria for testing documentation. The Implementation Correspondence Analysis section of this paper discusses the code correspondence performed by Ford Aerospace. The Security Model Task Report, Code Correspondence Analysis . . . also satisfies the intent of the Test Documentation criteria. [26 p. 154]*

6.4 Interaction

Interaction between the assessment team and the MGS developers had certain important characteristics. Perhaps the foremost characteristic was openness, in that the most accurate possible assessment resulted, and also that frequent feedback was provided to the developers. That feedback, in turn, resulted in a much better product than would have otherwise been possible both in terms of overall quality and also in terms of security assurance.

The interaction that existed was in a number of forms. It included formal review meetings, more informal development site meetings, delivery of interim and final products,

and direct access by customer and evaluators to internal information. This included off-line management and technical documentation and the development environment itself. The interaction also included many informal telephone conversations and electronic mail messages on a weekly and often daily basis.

Such open interaction is ideal and results in the most thorough possible evaluation with the least effort. Nevertheless, be aware that such openness is not guaranteed. Consequently, if an existing development contract does not require availability of informal documentation, one should be sure to establish ground rules for such availability. In this way, the developer will know that a successful evaluation is linked to the availability of informal documentation. This includes availability of documentation as well as verbal interaction.

6.5 Reporting

Some details of the assessment reporting for MGS are mentioned above. A final assessment report was produced that documented both the results of the preparation and observation (and, indirectly, interaction) and the results of the analysis. The inclusion of observation results (effectively a summary of the available information) is recommended. In that way readers of the assessment report are sure to have direct access to the same information.

The content of the MGS assessment report has three parts:

- results of observation (Sections I through VII);
- evaluation results (Sections VIII and IX); and
- detailed comments (Appendix A).

For the *results of observation*, the evaluators made a careful study of the provided documentation and made good use of opportunities to interview developers. They recorded, organized, and summarized the resulting relevant information in Sections I through VII. These sections include an introduction, a description of the environment and services provided, and descriptions of design documentation in the areas of system architecture, M-node architecture (mandatory access functionality), processor architecture, hardware architecture, software architecture, and protected resources. These sections also describe the MGS documentation in support of security assurance.

For the *evaluation results*, the evaluators examined the MGS as described in Sections I through VII and generated an evaluation in terms of areas specified in the TNI. They recorded the results of this in Sections VIII and IX. These sections include the TNI-based M-component evaluation and a conclusion, which summarizes the evaluation results. The conclusion includes the areas where the TNI requirements were met, areas needing more work, and specific concepts that should be emulated by other efforts.

The *detailed comments* provide direction for potential subsequent development and evaluation. The Appendix captures observations that might otherwise have been forgotten or ignored. The Appendix is the result of "bubbling up" information from the evaluation results and carefully recording the perceptions of the evaluators at the time. The appendix includes areas for positive recognition, areas for improvement, areas for further review, and observations about the TNI.

It should be noted that although the assessment report does not describe all the phases of assessment, the evaluation team did indeed proceed through all the phases. It is to be expected that certain phases, such as Preparation, would not appear in an assessment report.

7 ACRONYMS

CCA	code correspondence analysis
CCB	configuration congrol board *or* configuration change board
CDR	critical design review
CM	configuration management
CMS	code management system
COBOL	common business oriented language
COTS	commercial off-the-shelf
CSCI	computer software configuration item
CSC	computer software component
CSU	computer software unit
DID	data item description
DR	discrepancy report
DTLS	descriptive top-level specification
ICA	implementation correspondence analysis
ITSEC	information technology security evaluation criteria
MGS	multinet gateway system
NRL	Naval Research Laboratory
PDL	program design language
QA	quality assurance
SCCS	source code control system
SDF	software development folder
SDP	software development plan
SDR	software discrepancy report
STR	software trouble report
TCB	trusted computing base
TCSEC	Trusted Computer System Evaluation Criteria
TNI	Trusted Network Interpretation of the TCSEC
UDF	unit development folder

8 GLOSSARY

Abstraction	A view of a problem that extracts the essential information relevant to a particular purpose and ignores the remainder of the [non-essential] information.
B3	A specific class within one of the four divisions (D, C, B, A) of the DoD Trusted Computer System Evaluation Criteria [23]. The class B3 TCB must satisfy the reference monitor requirements that it mediate all accesses of subjects to objects, be tamperproof, and be small enough to be subjected to analysis and tests. To this end, the TCB is structured to exclude code not essential to security policy enforcement, with significant system engineering during TCB design and implementation directed toward minimizing its complexity. A security administrator is supported, audit mechanisms are expanded to signal security-relevant events, and system recovery procedures are required. The system is highly resistant to penetration.
Implementation	A realization of an abstraction in more concrete terms; in particular, in terms of hardware, software, or both. An implementation is also described as a machine executable form of a program, or a form of a program that can be translated automatically to machine executable form.
Implementation phase	The portion of the development life cycle that starts with a successful completion of a critical design review (CDR), includes the production of source code, the successful unit testing, CSC integration and testing, CSCI testing and acceptance into the system integration and test phase. As identified in [28] it is the period of time in the software life cycle during which a software product is created from design documentation and debugged.
Integration	The process of combining software elements, hardware elements, or both, into an overall system.
Product	A hardware and/or software package that can be bought off the shelf and incorporated into a variety of systems [5].
System	A specific installation with a particular purpose and a known operational environment [5].
Trusted System	A system that has been certified against some trust criteria.
Work Product	A particular result from activity or phase within a development method. For this chapter, a work product is a result of the implementation phase. Examples include source code, unit test procedures, unit test results, configuration management reports and appropriate analysis documentation.

9 BIBLIOGRAPHY

1. Ada Quality and Style Guide. Software Productivity Consortium. Version 02.01.01. (Software Productivity Consortium, 1880 Camus Commons Drive, North, Reston, VA 22091.
2. Beizer, Boris. *Software Testing Techniques*. 2nd edition., Van Nostrand Reinhold. New York, N. Y., 1990.
3. Beizer, Boris. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold. New York, N.Y., 1984.
4. Card, David N. and R. L. Glass. *Measuring Software Design Qualtiy*. Prentice Hall, Englewood Cliffs, N.J., 1990.
5. Commission of the European Communities. *Information Technology Security Evaluation Criteria*. Luxembourg.
6. Connell, John L. and Linda Brice Shafer. *Structured Rapid Prototyping*. Yourdon Press, Englewood Cliffs, N. J. , 1989.
7. 2167A Defense System Software Development. DoD-STD-2167A, Department of Defense, Washington, D.C., February 29, 1988.
8. Draft Standard for Software Reviews and Audits (P108), IEEE Computer Society, New York:, 1988.
9. Froscher, Judith N. and Charles N. Payne, Jr. The Handbook for the Computer Security Certification of Trusted Systems, Proceedings of the MILCOM '93 (classified), San Diego, CA, October 1992.
10. Gilb, Tom and Susannah Finzi. *Principles of Software Engineering Management*. Addison Wesley Publishing Co., New York, NY, 1988.
11. Gilb, Tom. *Software Inspection*. Addison Wesley Publishing Co., New York, NY, 1993.
12. Grady, Robert B. and Deborah L. Caswell. *Software Metrics; Establishing a Company Wide Program*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1987.
13. Greger, J. Brain. *Function Point Analysis*. Prentice Hall, Englewood Cliffs, N.J., 1989.
14. Halstead, M. H. *Elements of Software Science*. Elsevier North-Holland, 1977.
15. Henry, Sallie, and Dennis Kafura. Software Structure Metrics Based on Information Flow, *IEEE Transactions on Software Engineering*, 7(5), pp. 510 - 518, September 1981.
16. Humphrey, Watts S. *Managing the Software Process*. Addison-Wesley Publishing Company. Reading, Mass, 1990.
17. Jones, Caper. *Applied Software Management: Assuring Productivity and Quality*. McGraw-Hill, N.Y., 1991.
18. Kemerer, Chris, and Shyam Chidamber. Towards a Metric Suite for Object-Oriented Design. *Proceedings of Object-Oriented Programming, Systems, Languages and Applications Conference*, October, 1991.
19. Kolewe, Ralph. Metrics in Object-Oriented Design and Programming. *Software Development*, p. 53 - 62, October 1993.

20. McCabe, T. J. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4) pp. 308-320, December 1976.

21. Meyers, Glenford J. A Controlled Experiment in Program Testing and Code Walkthrough/Inspections. *Communications of the ACM*, 21(9) pp.760-768, 1978.

22. Meyers, Glenford J. *The Art of Software Testing*. JohnWiley & Sons, New York, 1979.

23. National Computer Security Center. *Trusted Computer System Evaluation Criteria*. DOD 5200.28-STD, Department of Defense, December 1985.

24. National Computer Security Center. *Trusted Network Interpretation of the Trusted Computer System Evaluation Criteria*. NCSC-TG-005, Department of Defense, July 1987.

25. National Computer Security Center, *A Guide to Understanding Configuration Management.* NCSC-TG-006, March, 1988.

26. National Security Agency. Final Design Assessment Report of Ford Aerospace Corporation Multinet Gateway System Advanced Development Model, Version 4.0. Information Systems Security Organization. C22-REPT-01-90, Library No. S234,984.

27. Putnam, Lawrence and Ware Myers. *Measures for Excellence: Realiable Software on Time, within Budget*. Yourdon Press, Englewood Cliffs, N.J., 1992.

28. *Software Engineering Standards: Glossary of Software Engineering Terminology*. ANSI/IEEE Std 729-1983, IEEE, N.Y., N.Y., 1984.

29. Schultz, H. P. "Software Management Metrics." MITRE, ESD-R-88-001, May 1988.

30. Vickers Benzel, T.C. Developing Trusted Systems Using DOD-STD-2167A. *Proceedings of the Fifth Computer Security Applications Conference*. IEEE, N.Y., N.Y. December 1989, pp. 166 - 176.

31. Youll, David P. *Making Software Visible: Effective Project Control*. John Wiley & Sons. Chichester, 1990.